SMS USERS GUIDE

Tom Henderson

Dan Schaffer

Mark Govett

Leslie Hart

Advanced Computing Branch
Aviation Division
NOAA/Forecast Systems Laboratory
325 Broadway
Boulder, Colorado 80303

May 2000
SMS Software Version: 2.1

http://www-ad.fsl.noaa.gov/ac/sms.html.

1 Introduction

This document describes how the Scalable Modeling System's (SMS) directives can be used to parallelize a serial Fortran program for distributed or shared memory machines. SMS is intended for use with programs that perform computations on regular gridded data sets. The primary application area thus far has been Numerical Weather Prediction (NWP) models. SMS has been used to parallelize NWP models that use finite difference approximation (FDA) or the spectral transform method. SMS is general enough that it should be useful for parallelizing similar programs in other application areas.

Before reading this document, the reader should first read the companion overview document "SMS: A Directive-Based Parallelization Tool for Shared and Distributed Memory High Performance Computers". It is assumed that the reader of this Users Guide is familiar with the concepts and terms introduced in the overview document. The reader should also be familiar with basic parallel processing concepts such as distributed and shared memory, message latency and bandwidth, the Single Program Multiple Data (SPMD) programming model, and dependence analysis. The overview document describes these concepts briefly and contains references for further reading. After reading this Users Guide, the reader should have a good understanding of the steps that need to be taken to parallelize a serial program using the SMS directives. If more detailed information about any directive is needed, the reader should refer to the companion reference document, "SMS Reference Manual". Answers to common questions and detailed discussions of problems not covered here may be found on the SMS FAQ web site at:

http://www-ad.fsl.noaa.gov/ac/SMS_FAQ.html

1.1 Organization of this Document

The SMS Users Guide begins by introducing the SMS directives in their simplest form. Section 2 introduces the most fundamental SMS directives with simple example programs that use the method of finite difference approximation. This section also introduces other SMS directives that are useful in transform-based programs such as spectral NWP models. The remaining sections describe in detail how the SMS directives are used in more complex situations. Section 3 explains how to divide work among multiple processes by the method of data decomposition and how to parallelize loops. Additional loop index translations needed during parallelization are described in Section 4. Special directives that provide direct control over code translation are introduced in Section 8. Sections 5, 6 and 7 cover further details about the inter-process communication directives introduced in Section 2. Section 9 describes parallel I/O. Directives that control program termination are dealt with in Section 10. Sections 11 and 12 explain how to build and run parallel SMS programs.

1.2 Terms and Conventions

Throughout most of this document, the term "process" is used instead of "processor" or "CPU". "Process" is slightly more general because it is possible to run more than one process on a single "processor" (and this may actually make sense on some types of CPU's that provide direct hardware support for multi-threaded applications). However, on most machines there will be a one-to-one mapping of processes to processors.

Fortran source code will appear in `courier` font. When program identifiers appear inside the main body of text, they will also be ***`italicized`***. Large blocks of code will include line numbers to simplify discussions. Commands will also appear in `courier` font and will be preceded by a generic command line prompt, ">>". The results of commands will appear in `courier` font as well. Warning messages output by SMS will be **`courier bold`**. File names will appear in *italics* when not in code examples or command lines. SMS directives will appear in **`bold`** in code examples. When directive parameters appear in the text they will be ***`courier font, bold and italicized`***. Sometimes example code will be a slightly modified version of a previous example. In that case, the changed lines will be highlighted.

2 Getting Started

2.1 Basic Parallelization Steps

The first step in any parallelization effort is to understand the performance characteristics of the serial program. Program components that take little time to run may not need to be parallelized at all. Next, dependence analysis is performed to identify the places in the code where inter-process communication may be required. Dependencies will be discussed as relevant SMS directives are introduced. A strategy for dividing the work among the processes must then be chosen. SMS uses the method of domain decomposition in which portions of large arrays, and their associated computations, are assigned to each process. The dependence analysis is used to help pick optimal decompositions that will minimize inter-process communication. Finally, SMS directives are added to parallelize the code.

To build the parallel code, the Parallelizing Pre-Processor (PPP) is first run to translate the code with directives into new parallel source code. The translated code is then compiled and linked with SMS libraries to produce an executable program that can be run on multiple processes. The smsRun command is used to run the parallel program.

PPP supports many common extensions to ANSI standard Fortran77, as will be seen in the code examples that follow. A few Fortran90 language features (such as full array assignment) are also supported. Other language extensions supported include namelist, pointer, include, do-enddo, automatic arrays, and while statements. A more detailed description of supported language features can be found at the following SMS site:

http://www-ad.fsl.noaa.gov/ac/SMS_Supported_Fortran_Features.html

2.2 A Very Simple Program

Below is a simple Fortran program that prints a message on the screen:

```
program basic_ex1
print *,'HELLO'
end
```

If this program were stored in a file named *basic_ex1.f*, it could be built using the following command:

```
>> f77 -o basic_ex1 basic_ex1.f
```

The above command assumes that the Fortran compiler is named "f77". When run, the program produces the expected output on the screen:

```
>> basic_ex1

 HELLO
```

This program is simple enough that a parallel version can be built directly without adding any SMS directives. To build with SMS, first run the Parallel Pre-Processor (PPP) to convert the print statements into parallel print statements:

```
>> ppp basic_ex1.f
```

The above command assumes that the SMS environment variable has been correctly set and that $SMS*/bin* is in the current path. For example, if SMS is installed in directory */usr/local/sms/* then (assuming a c-shell environment) the SMS environment variable should be set as follows:

```
>> setenv SMS /usr/local/sms
```

The path could be modified using a command like this:

```
>> set path= ( $SMS/bin $path )
```

See Section 12.3 for details about setting other environment variables used by SMS. PPP translates the serial code in *basic_ex1.f* into parallel code and places the result in file *basic_ex1_sms.f*. Depending on the configuration of PPP, other temporary files may also be created. The next step is to compile *basic_ex1_sms.f* and link it to the SMS libraries.

```
>> f77 -c -I $SMS/include basic_ex1_sms.f
>> f77 -o basic_ex1_sms -I $SMS/include basic_ex1_sms.o -L $SMS/lib \
   -lppp_support -lfnnt -lnnt -lsrs -lmpi
```

The above example assumes common behavior for f77 options "-I" (specify path for include files) and "-L" (specify path for libraries). Some Fortran compilers handle these options in slightly different ways. Note that link argument "-lmpi" links to the Message Passing Interface (MPI) library. SMS uses MPI to perform underlying low-level inter-process communication on most supported machines. Some machines may require different linkers or linker arguments to link to their MPI libraries.

The next step is to run the parallel program:

```
>> smsRun 1 basic_ex1_sms
```

The smsRun command shown above runs program *basic_ex1_sms* on 1 process. The output written to the screen will look something like this:

```
SMS:: Program started: 1999:12:02::15:55:22
SMS:  BITWISE EXACT reductions will NOT be used.
 HELLO
SMS:: Program complete, exiting: 1999:12:02::15:55:22 Elapsed Time = 0 sec.
```

The text lines beginning with "SMS::" are time-stamps printed by SMS when a program begins and when it ends. These time-stamps are a useful guide for measuring wall-clock run times. The second text line is another message from SMS that indicates default behavior of some reduction operations discussed in Section 7.2. From now on, these diagnostic messages from SMS will usually be omitted for brevity. The remaining line contains the text we already saw when this program was run as a serial Fortran code.

The program can be run on 3 processes using the smsRun command like this:

```
>> smsRun 3 basic_ex1_sms
```

The following text appears on the screen:

```
 HELLO
```

This looks just like the run made on one process. Why? By default, SMS prints only one message per Fortran print (or write) statement to mimic the behavior of the original serial code as closely as possible. SMS also provides other "parallel print" modes, as described later in this section and in detail in Section 9.3.

2.3 Simple Computation on a Regular Grid

Example 2-1 illustrates a very simple code that initializes an array, performs a simple computation, and prints results on the screen. It consists of two parts: include file *basic.inc* and source file *basic_ex2.f*.

```
[Include file:  basic.inc]

      integer im, jm
      common /sizes_com/ im, jm


[Source file:  basic_ex2.f]

      program basic_ex2
      include 'basic.inc'
      im = 10
      jm = 10
      call compute
      end

      subroutine compute
      include 'basic.inc'
      integer i, j, xsum
      integer x(im,jm)
      do 100 j=1,jm
      do 100 i=1,im
        x(i,j) = 1
```

```
  100 continue
      xsum = 0
      do 200 j=1,jm
      do 200 i=1,im
        xsum = xsum + x(i,j)
  200 continue
      print *,'xsum = ',xsum
      return
      end
```

Example 2-1: A simple serial code to initialize an array and print a global sum.

This program initializes array *x*, sums the elements of *x*, and prints the result on the screen as shown below:

```
>> basic_ex2
 xsum = 100
```

Notice that this program uses automatic (dynamically allocated) arrays instead of traditional Fortran77 static array declarations. This technique of dynamic memory allocation is a widely supported extension to standard Fortran77. The SMS directives support both dynamic and static memory allocation schemes. Examples with dynamic memory allocation are shown first because they are slightly simpler. Static allocation examples appear in Section 3.3.

2.3.1 Parallelization by Domain Decomposition

Programs such as this one that involve computations on regular grids are often best parallelized using the method of domain decomposition. Arrays and the computations performed on them are "decomposed" (divided up) among the processes as evenly as possible. For example, Figure 2-1, Figure 2-2, and Figure 2-3 show how array *x* might be decomposed in the *i* dimension over one, two and three processes.

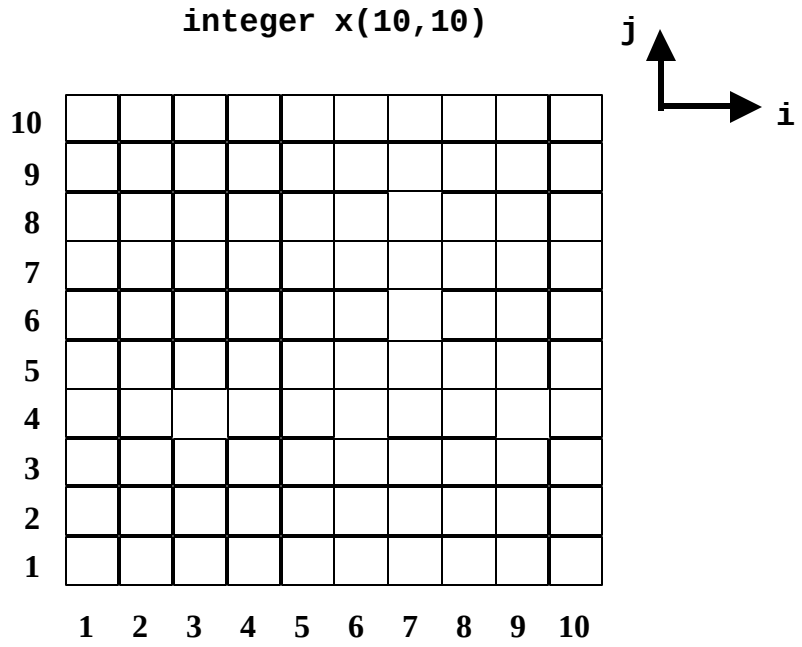


Figure 2-1: The graphical representation of a non-decomposed 10 by 10 integer array.

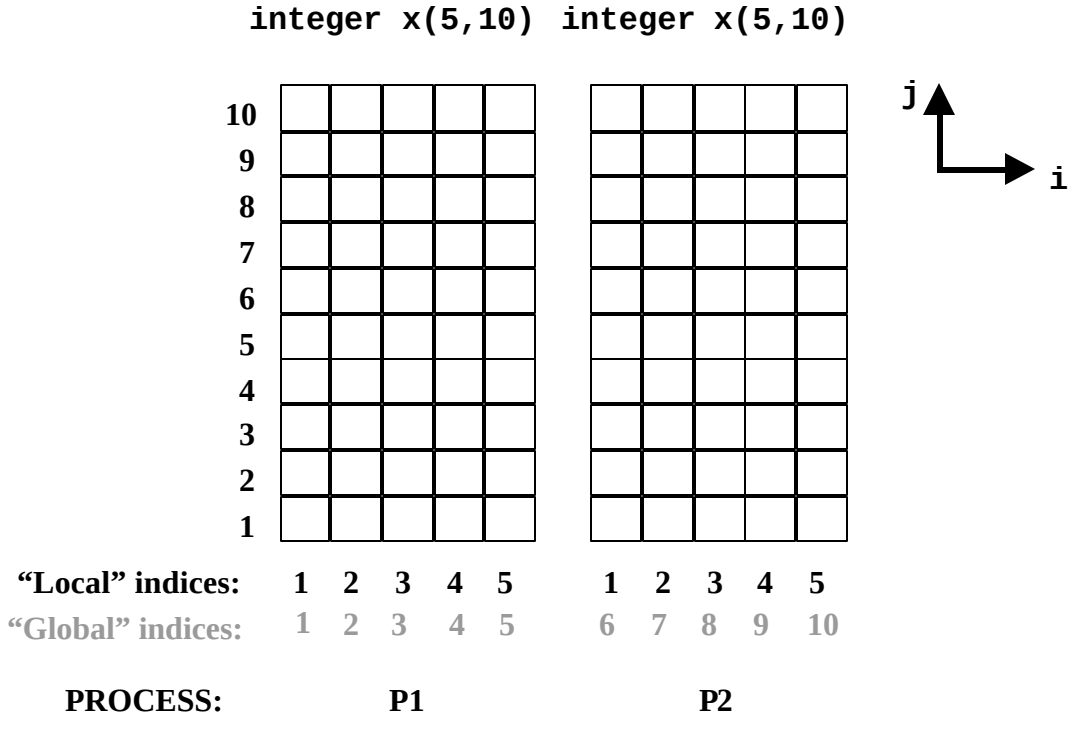


Figure 2-2: An illustration of a 10 by 10 array decomposed over two processes. These integer arrays are now local arrays declared by each process. Local addressing is used to access array elements.

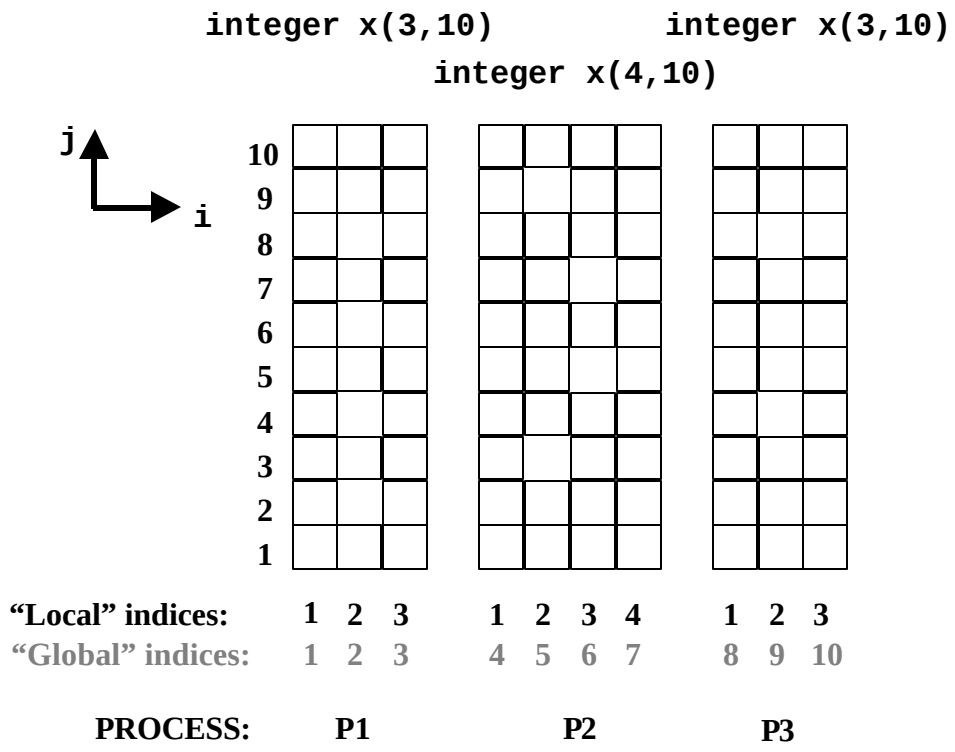


Figure 2-3: A 10 by 10 array decomposed over three processes. In this example, the locally declared size of process P2 is larger than the sizes of P1 or P3.

Note that the sub-domains of array *x* become smaller as the number of processes increases. These sub-domains are referred to as "local" arrays because they cannot be accessed by other processes on a distributed memory machine. In SMS terms, the original array *x* in the serial code is sometimes referred to as a "global array". Indices used to access a global array are called "global indices" while indices used to access a local array are called "local indices". Similarly, sizes of the dimensions of a global array are called "global sizes" and sizes of the dimensions of a local array are called "local sizes". SMS treats memory as if it were distributed because this works on machines with either shared or distributed memory.

In this program, domain decomposition of array *x* requires three basic steps. First, the way in which *x* will be decomposed must be described. For this simple example, we choose to decompose only in the *i* dimension. (Decompositions of two dimensions are discussed in Section 3.2). Second, the declarations of array *x* should be modified to reflect smaller local sizes. Finally, the start and stop indices of each relevant loop must be changed to reflect the smaller range of local indices. These three steps are accomplished using four SMS directives. The DECLARE_DECOMP and CREATE_DECOMP directives are used to describe a decomposition. Array declarations are modified using the DISTRIBUTE directive while loop start and stop indices are changed using the PARALLEL directive. These directives have been inserted into the serial program as shown in Example 2-2 :

```
[Include file:  basic.inc]
 1        integer im, jm
 2        common /sizes_com/ im, jm
 3  CSMS$DECLARE_DECOMP(DECOMP_I)

[Source file:  basic_ex2.f]

 1        program basic_ex2
 2        include 'basic.inc'
 3        im = 10
 4        jm = 10
 5  CSMS$CREATE_DECOMP(DECOMP_I, <im>, <0>)
 6        call compute
 7        end
 8
 9        subroutine compute
10        include 'basic.inc'
11        integer i, j, xsum
12  CSMS$DISTRIBUTE(DECOMP_I, <im>) BEGIN
13        integer x(im,jm)
14  CSMS$DISTRIBUTE END
15  CSMS$PARALLEL(DECOMP_I,<i>) BEGIN
16        do 100 j=1,jm
17        do 100 i=1,im
18          x(i,j) = 1
19    100 continue
20        xsum = 0
21        do 200 j=1,jm
22        do 200 i=1,im
23          xsum = xsum + x(i,j)
24    200 continue
25  CSMS$PARALLEL END
26        print *,'xsum = ',xsum
27        return
28        end
```

Example 2-2: A simple serial code with comment-based SMS directives added.

Notice that each of the SMS directives begins with five characters "CSMS$" which makes it a Fortran comment. This is true for all SMS directives. The advantage of using comment-based directives is that the original serial program can still be built and run after directives are added.

Also, note that both the DISTRIBUTE and PARALLEL directives come as BEGIN-END pairs. When an SMS directive appears in this form, its scope consists of all lines of code between the "BEGIN" and "END" directives. Some SMS directives, such as TRANSFER (Section 6) and REDUCE (Section 7) may be used either alone or as a BEGIN-END pair. The text translation effects of a BEGIN-END directive pair do not extend into called subroutines.

The first directive, DECLARE_DECOMP, is used to give a name to the SMS decomposition that will be used to divide among the processes the work done in loops 100 and 200. In this DECLARE_DECOMP directive the single parameter, ***DECOMP_I***, is the user-chosen name for

the decomposition. Any valid Fortran variable name (up to 20 characters long) may be used to name a decomposition provided it does not conflict with any variable in the serial code.

Next, the CREATE_DECOMP directive is used to describe what kind of decomposition ***DECOMP_I*** will be. The first parameter is the decomposition name ***DECOMP_I*** specified in the DECLARE_DECOMP directive. The second parameter, ***<im>***, describes the decomposition as a 1-dimensional decomposition where the number of data points in the original serial dimension (the global size) is ***im***. The last parameter, ***<0>***, indicates that this decomposition will have no halo regions (halo thickness = 0). Halo regions are introduced later in this section and are described in detail in Section 5.1.

The third directive, DISTRIBUTE, associates arrays with decompositions. The second parameter is used to indicate how array dimension(s) correspond to the dimensions of the decomposition named ***DECOMP_I***. In this simple one-dimensional decomposition, ***<im>*** indicates that all array dimensions of size ***im*** will be decomposed as described by the single dimension of the SMS decomposition named ***DECOMP_I***. The distinction between "dimension of an array" and "dimension of an SMS decomposition" will become more clear in the two-dimensional decomposition examples shown later in Section 3.2.

The DISTRIBUTE directive does two things. First, it identifies array declarations that will be translated to use local sizes. In the above example program, the DISTRIBUTE directive will cause PPP to translate the declaration of ***x*** to the local declarations shown in Figure 2-1, Figure 2-2, and Figure 2-3. The second task of DISTRIBUTE is to provide information about how each array is decomposed to other SMS directives and to support automatic parallelization of binary I/O. These features are described in detail in later sections.

Finally, the PARALLEL directive identifies loops that must be modified to span the smaller local arrays during translation. The second parameter, ***<i>***, indicates that loops with loop index ***i*** should be translated to span the decomposed dimension of array ***x***. For example, if the program in Example 2-1 is run on two processes then ***i*** loops 100 and 200 will span local indices 1 through 5 on each process. A second function of the PARALLEL directive is to provide other enclosed directives with a "default" SMS decomposition. Directives such as TO_GLOBAL, TO_LOCAL, GLOBAL_INDEX, and HALO_COMP can all determine the current SMS decomposition from an enclosing PARALLEL directive. Thus, it is not necessary to use a decomposition name in these directives when they appear inside a PARALLEL directive. These directives are described in more detail in later sections.

Building this code is a bit more complicated than the previous example due to the presence of the include file that contains a directive. Two commands are now needed. The first translates the include file:

```
>> ppp --header basic.inc
```

The "--header" option to the PPP command indicates that the file is an include file and must be translated differently than a standard Fortran source file. In the command above, include file *basic.inc* will be translated into new SMS include file *basic.inc.SMS*. The second command requires PPP option "--Finclude" to translate the Fortran source file:

```
>> ppp --Finclude=basic.inc basic_ex2.f
```

The "--Finclude" option to the PPP command indicates that file *basic.inc* is an include file that has been translated by PPP. During translation of source file *basic_ex2.f*, any lines that include this file will be translated from

```
      include 'basic.inc'
```

to

```
      include 'basic.inc.SMS'
```

to ensure that the translated include file is used.

Running this program on one process produces the expected result.

```
>> smsRun 1 basic_ex2_sms
 xsum =  100
```

However, when this program is run on two and three processes, the values of xsum differ from the serial run.

```
>> smsRun 2 basic_ex2_sms
 xsum =  50

>> smsRun 3 basic_ex2_sms
 xsum =  30
```

Why did the parallel program produce incorrect results? The answer lies in the computations made in loop 200. In this loop, all of the elements of array *x* are summed and the result is placed in variable *xsum*. However, when the program is run on two or three processes, each process sums only its own local sub-domain of *x* as illustrated in Figure 2-4, and Figure 2-5. To get a global result, we will need an additional directive that will be introduced later in this section.

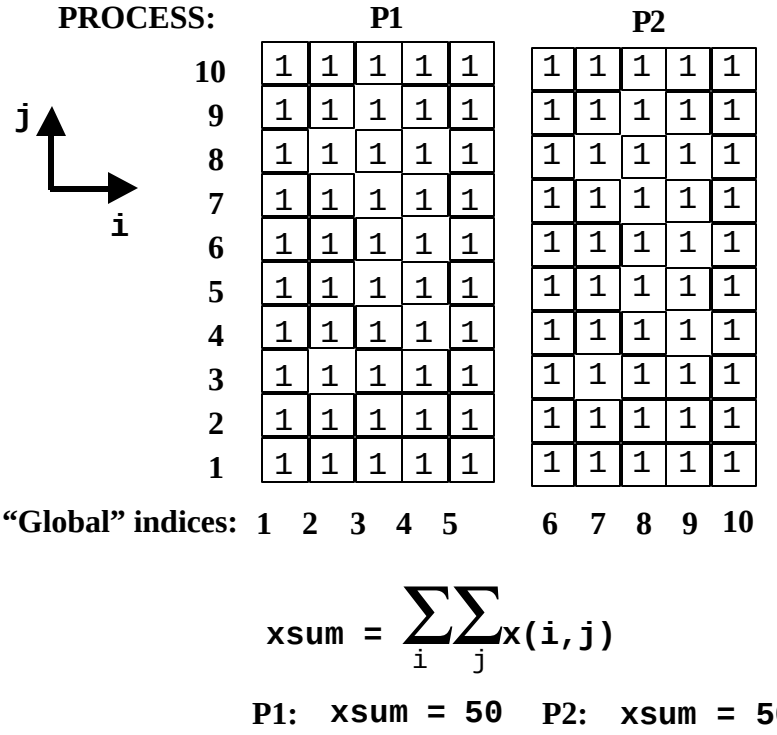


Figure 2-4: Each process sums their local portion of the array x.

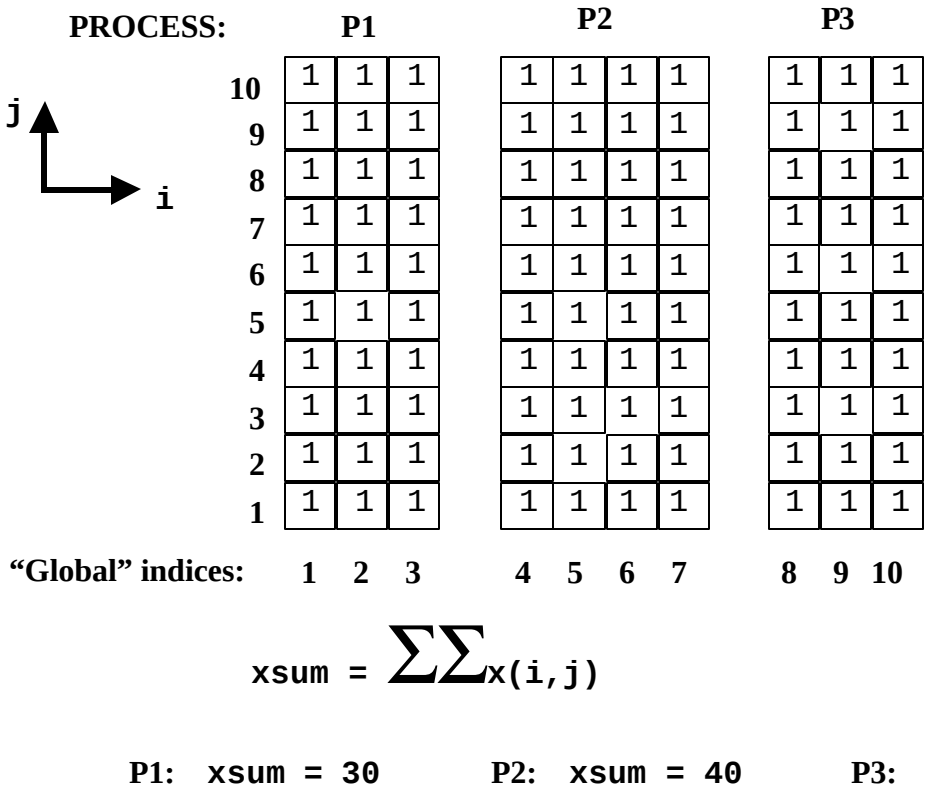


Figure 2-5: In this example, local sums are produced on each of the three processes.

2.3.2 Parallel Printing

By default, only one process will print a message when a print statement is encountered. Therefore, the value of *xsum* printed is the value of *xsum* computed locally only on the printing process. We can see the value of *xsum* on every process by changing the default print behavior with the PRINT_MODE directive. The print statement in the above program would be modified as shown below:

```
CSMS$PRINT_MODE(ASYNC) BEGIN
      print *,'xsum = ',xsum
CSMS$PRINT_MODE END
```

This PRINT_MODE directive changes the print mode from the default mode to "asynchronous" mode. When a print statement is encountered in asynchronous print mode, each process will print a message to the screen. When run on two processes, the following results are printed:

```
>> smsRun 2 basic_ex2_sms
 xsum =  50
 xsum =  50
```

Clearly, each process has computed the correct sum for its local half of array *x*. When run on three processes we may see any of the following results:

```
>> smsRun 3 basic_ex2_sms
 xsum =  40
 xsum =  30
 xsum =  30

>> smsRun 3 basic_ex2_sms
 xsum =  30
 xsum =  40
 xsum =  30

>> smsRun 3 basic_ex2_sms
 xsum =  30
 xsum =  30
 xsum =  40
```

In the asynchronous print mode, the messages printed by each process may come out in any order. Another parallel print mode supported by SMS is the "ORDERED" print mode does preserve process order. Section 9.3 describes the SMS print modes in more detail.

2.3.3 Reduction

We have seen that each process has computed the correct sum for its local sub-domain of array *x*. To generate the same result as the original serial code, these local sums must be added

together as shown in Figure 2-6 and Figure 2-7. In more general terms, the computed value of *xsum* depends on all of the values of array *x*. This is known as a "global dependence" because the result of the computation depends on every element of global array *x*.

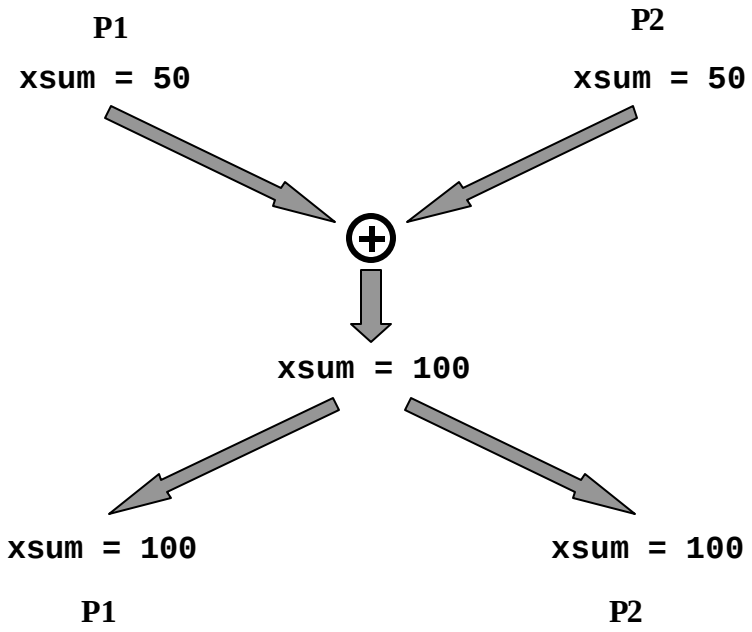


Figure 2-6: In this example, the reduction gathers the local sums, computes a global sum and then broadcasts the result out to the processes.

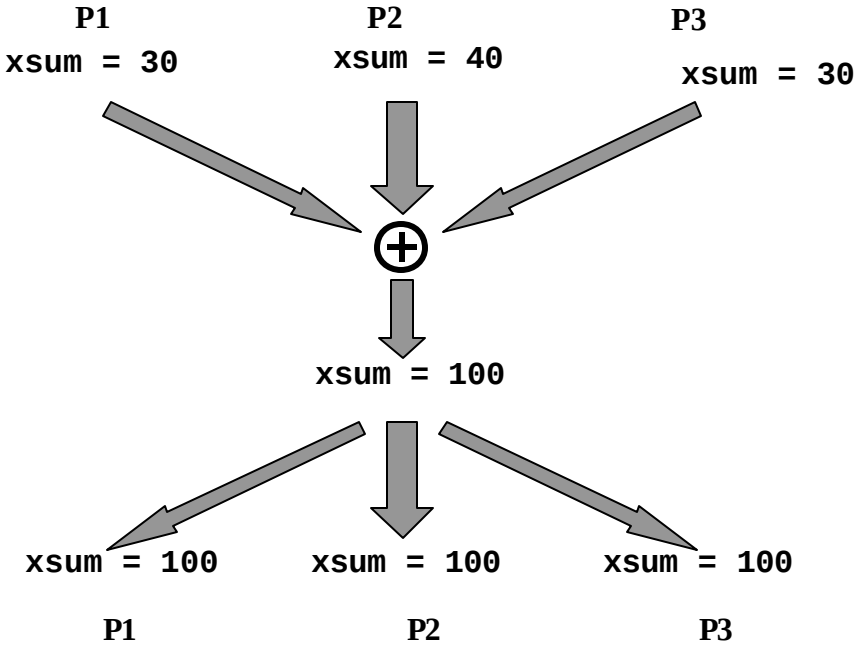


Figure 2-7: A reduction performed on three processes produces a global sum of 100 on every process.

The REDUCE directive is used to resolve this dependence. To use the REDUCE directive, insert the following line immediately before the print statement on line 26 of Example 2-2:

```
CSMS$REDUCE(xsum,SUM)
```

The REDUCE directive performs communications necessary to reduce the local values of a variable on each process to a single value that is identical on all processes. A specified operator is used to combine the values from each process. The first parameter indicates that *xsum* is the name of the variable to be reduced. The second parameter, ***SUM***, specifies that the local values of *xsum* will be summed during reduction. Reductions are described in more detail in Section 7. The parallel program now produces the expected results when run on various numbers of processes:

```
>> smsRun 2 basic_ex2_sms
 xsum =  100
>> smsRun 3 basic_ex2_sms
 xsum =  100
```

2.4 Boundary Initialization

In Example 2-2 (page 13), all elements of array *x* were initialized to the same value. Often, it is desirable to initialize array elements differently depending on their location. This occurs often in NWP models where elements near the model boundaries may be treated differently than other array elements. For example, the following variant of subroutine *compute* in Example 2-2 sets elements on the array boundaries where *i=1* or *i=im* to 2 and all other elements to 1 as illustrated in Figure 2-8, and specified in Example 2-3.

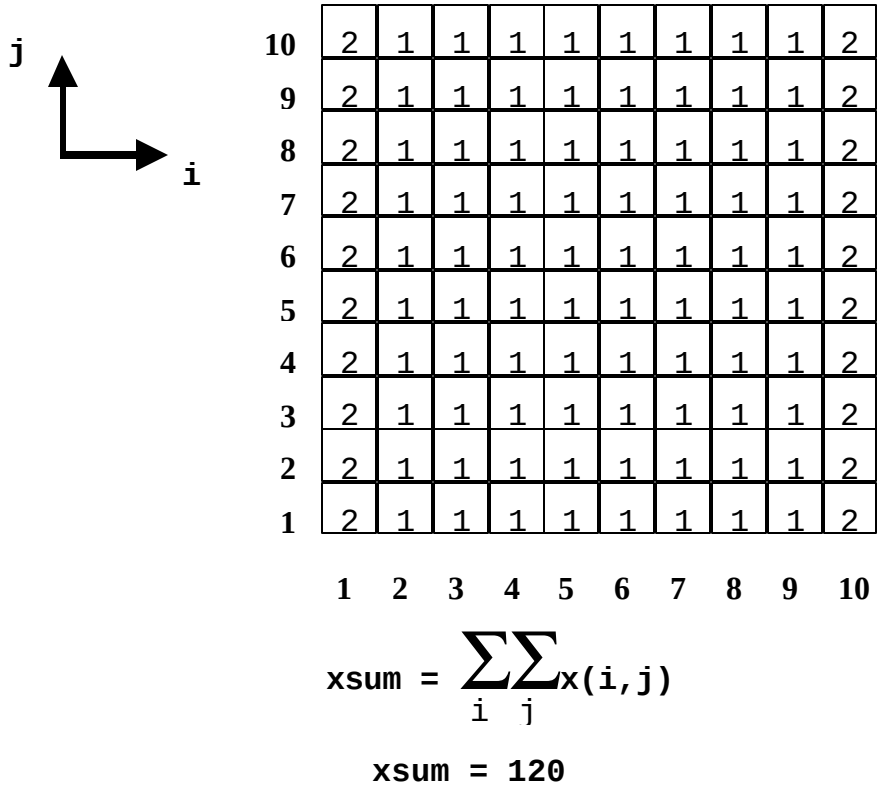


Figure 2-8: An illustration of a boundary initialization where edge point values are different than interior points.

```
 1        subroutine compute
 2        include 'basic.inc'
 3        integer i, j, xsum
 4  CSMS$DISTRIBUTE(DECOMP_I, <im>) BEGIN
 5        integer x(im,jm)
 6  CSMS$DISTRIBUTE END
 7  CSMS$PARALLEL(DECOMP_I,<i>) BEGIN
 8        do 100 j=1,jm
 9        do 100 i=1,im
10          x(i,j) = 1
11    100 continue
12        do 110 j=1,jm
13          x( 1,j) = 2
14          x(im,j) = 2
15    110 continue
16        xsum = 0
17        do 200 j=1,jm
18        do 200 i=1,im
19          xsum = xsum + x(i,j)
20    200 continue
```

```
21  CSMS$PARALLEL END
22  CSMS$REDUCE(xsum,SUM)
23        print *,'xsum = ',xsum
24        return
25        end
```

Example 2-3: Boundary Initialization requires special handling.

When the serial version of Example 2-3 is run, the following results are printed on the screen:

```
>> basic_ex3
 xsum =  120
```

However, when the parallel code is run on more than one process, results are unpredictable:

```
>> smsRun 2 basic_ex3_sms
 xsum =  138
>> smsRun 3 basic_ex3_sms
<core dump>
```

The reason for these erroneous results can be seen by examining new loop 110 in detail. Line 14 in loop 110 contains the following statement:

```
        x(im,j) = 2
```

This statement will perform the following assignments:

```
        x(10, 1) = 2
        x(10, 2) = 2
...
        x(10,10) = 2
```

However, on two processes, each sub-domain of array *x* has local size *x(5,10)* (see Figure 2-2) so *x(10,10)* is out of bounds. In fact, this statement will cause an out-of-bounds assignment during any run on two or more processes. The behavior of any program that performs such assignments is unpredictable.

The statement on line 13 also causes incorrect results, even though it does not do out-of-bounds assignment:

```
        x( 1,j) = 2
```

This statement will perform the following assignments:

```
        x(1, 1) = 2
        x(1, 2) = 2
...
        x(1,10) = 2
```

However, these assignments will not produce the desired results when two or more processes are used because the index in the ***i*** dimension ("1") is a global index. The effects of this erroneous assignment statement are shown in Figure 2-9 and Figure 2-10.

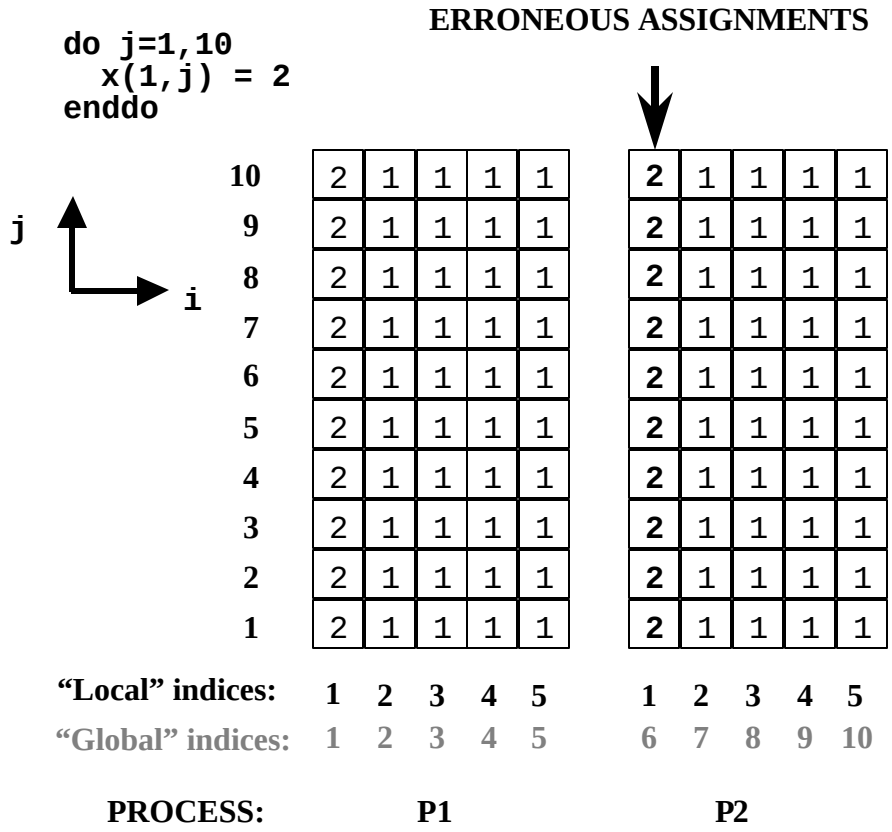


Figure 2-9: Boundary initialization of decomposed data require special handling to avoid erroneous assignments on local index 1 by process P2.

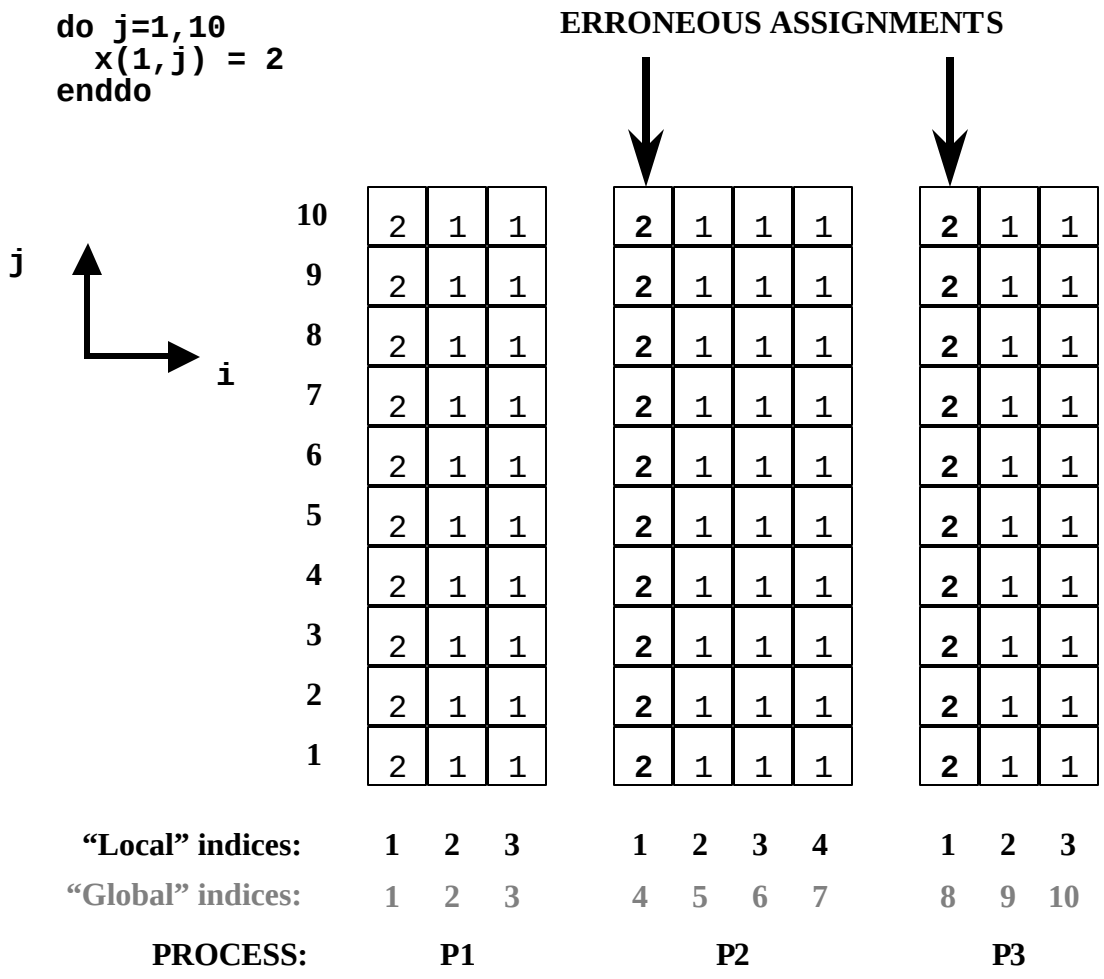


Figure 2-10: Boundary initialization of global index 1 will cause erroneous assignments in the local arrays on process P2 and P3.

Two problems must be solved to repair this code. First, the global indices ***1*** and ***im*** must be translated to their local equivalents. Second, the assignment statements must be modified so they are only executed on the processes that contain the specified global indices in their local sub-domains. The GLOBAL_INDEX directive solves these problems as shown below:

```
      do 110 j=1,jm
CSMS$GLOBAL_INDEX(1) BEGIN
        x( 1,j) = 2
        x(im,j) = 2
CSMS$GLOBAL_INDEX END
  110 continue
```

The GLOBAL_INDEX directives perform the correct index translations and ensure that the enclosed statements are only executed on the appropriate processes. The parameter in the GLOBAL_INDEX directive, ***"1"***, indicates that these translations will be applied to array indices that correspond to the first (and in this case only) decomposed dimension. In this case, the decomposed dimension corresponds to the ***i*** dimension of array ***x***. (The concept of "decomposed dimension" is explained in detail in Section 3.) The effects of the GLOBAL_INDEX directives on the assignments of ***x(1,j)*** and ***x(im,j)*** are shown for the two process case in Figure 2-11.

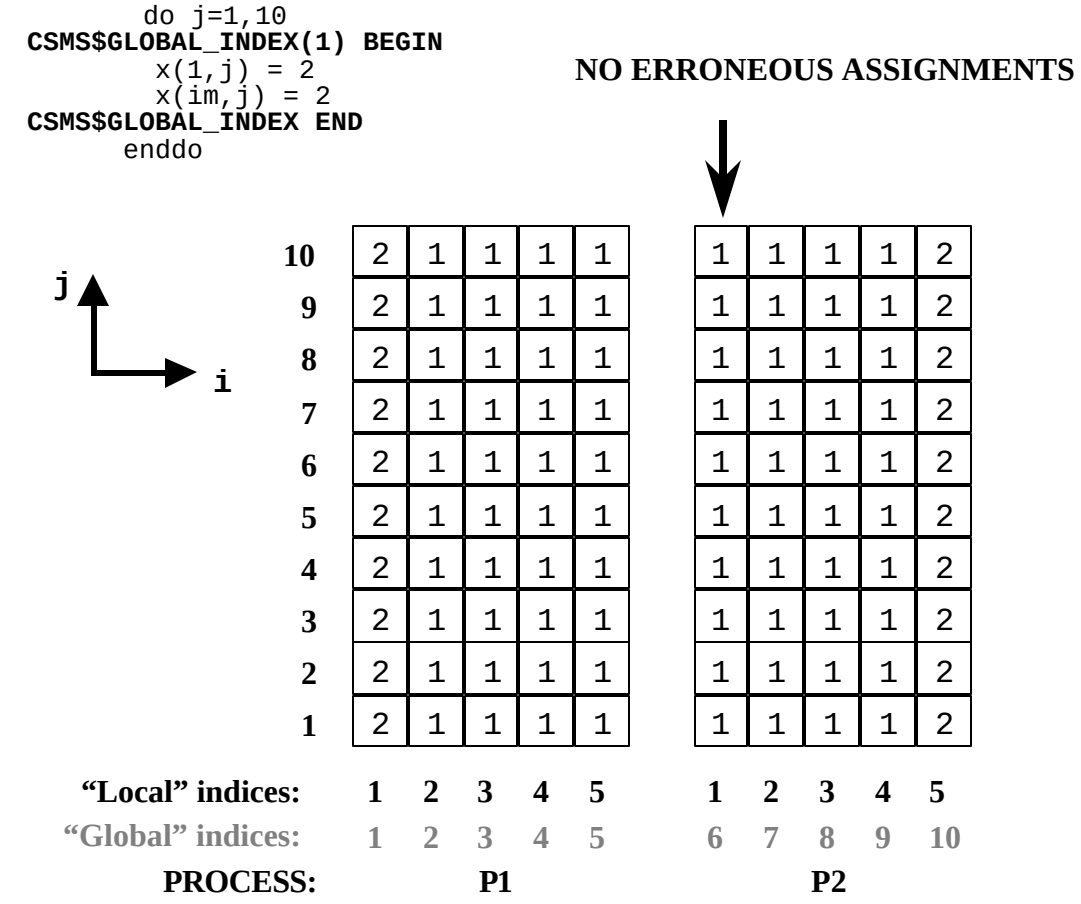


Figure 2-11: GLOBAL_INDEX is used to correctly initialize the boundaries of the array *x*.

Now when the parallel code is run, results match the serial code:

```
>> smsRun 2 basic_ex3_sms
 xsum =  120
>> smsRun 3 basic_ex3_sms
 xsum =  120
```

2.5 A Simple FDA Program

The following example is a FDA program that solves Laplace's equation on a two-dimensional surface with fixed boundaries using Jacobi relaxation. On a two-dimensional surface, Laplace's equation takes the form:

$$\frac{\partial^2 f}{\partial^2 x} + \frac{\partial^2 f}{\partial^2 y} = 0$$

A simple approach is to discretize the two-dimensional space and use a finite difference approximation to the derivatives to seek a numerical solution. The discrete equation is:

```
4*f(i,j) - f(i-1,j) - f(i+1,j) - f(i,j-1) - f(i,j+1) = 0
```

The initial state is *f* on the boundaries. The boundaries are constant and non-periodic. The above equation is solved for *f(i,j)* iteratively until it converges. The solution is said to converge when the difference between successive solutions is less than a specified threshold. The difference between values of *f(i,j)* in two successive iterations is the following:

```
df(i,j) = (1/4) * (f(i-1,j) + f(i+1,j) + f(i,j-1) + f(i,j+1)) - f(i,j)
```

Using the method of Jacobi relaxation, the value of *f(i,j)* during an iteration is calculated fom the value of *f(i,j)* computed in the previous iteration as follows:

```
fnew(i,j) = fold(i,j) + df(i,j)
```

In Example 2-4 below, boundary elements of array *f* are initially set to 2.0 (lines 25-31). Laplace's equation is then solved and diagnostic messages are printed on the screen. Previously described SMS directives have already been inserted.

```
[Source file:  laplace.f]
 1        program laplace
 2        include 'basic.inc'
 3        im = 10
 4        jm = 10
 5  CSMS$CREATE_DECOMP(DECOMP_I, <im>, <0>)
 6        call laplace
 7        end
 8
 9        subroutine laplace
10        include 'basic.inc'
11        integer i, j, iter
12        real max_error
13        real tolerance
14        parameter (tolerance = 0.001)
15  CSMS$DISTRIBUTE(DECOMP_I, <im>) BEGIN
16        real f(im,jm), df(im,jm)
```

```
17  CSMS$DISTRIBUTE END
18  CSMS$PARALLEL(DECOMP_I,<i>) BEGIN
19        do 100 j=1,jm
20        do 100 i=1,im
21          f(i,j) = 0.0
22    100 continue
23        do 110 j=1,jm
24  CSMS$GLOBAL_INDEX(1) BEGIN
25          f( 1,j) = 2.0
26          f(im,j) = 2.0
27  CSMS$GLOBAL_INDEX END
28    110 continue
29        do 120 i=1,im
30          f(i, 1) = 2.0
31          f(i,jm) = 2.0
32    120 continue
33        iter = 0
34        max_error = 2.0 * tolerance
35  C main iteration loop...
36        do while ((max_error .gt. tolerance) .and. (iter .lt. 1000))
37          iter = iter + 1
38          max_error = 0.0
39          do 200 j=2,jm-1
40          do 200 i=2,im-1
41            df(i,j) = 0.25*(f(i-1,j) + f(i+1,j) + f(i,j-1) + f(i,j+1))
42       &                - f(i,j)
43    200 continue
44          do 300 j=2,jm-1
45          do 300 i=2,im-1
46            if (max_error .lt. abs(df(i,j))) then
47              max_error = abs(df(i,j))
48            endif
49    300 continue
50  CSMS$REDUCE(max_error, MAX)
51          do 400 j=2,jm-1
52          do 400 i=2,im-1
53            f(i,j) = f(i,j) + df(i,j)
54    400 continue
55        enddo
56  CSMS$PARALLEL END
57        print *, 'Solution required ',iter,' iterations'
58        print *, 'Final error = ', max_error
59
60        return
61        end
```

Example 2-4: Serial code plus directives illustrate a parallel solution to Laplace's equation. This solution, using a one-dimensional decomposition, produces incorrect results.

Notice that the REDUCE directive uses the maximum operator to reduce *max_error* via parameter ***MAX***. The Jacobi relaxation will also work if average error is used instead of maximum error. However, using maximum error guarantees bit-wise exact results as described in Section 7.2.

When the serial program is run, the following messages are printed on the screen:

```
>> laplace
 Solution required  85  iterations
 Final error =  9.9968910E-4
```

When the parallel program is run on more than one process, results are incorrect:

```
>> smsRun 2 laplace_sms
 Solution required  45  iterations
 Final error =  9.9253654E-4

>> smsRun 3 laplace_sms
 Solution required  131  iterations
 Final error =  9.9420547E-4
```

Why do results change for different numbers of processes? The answer lies in the computations made on lines 41 and 42:

```
df(i,j) = 0.25*(f(i-1,j) + f(i+1,j) + f(i,j-1) + f(i,j+1)) - f(i,j)
```

Here, each ***df(i,j)*** is computed from ***f(i-1,j)***, ***f(i+1,j)***, ***f(i,j-1)***, f***(i,j+1)***, and ***f(i,j)***. This type of dependence is called an "adjacent dependence" because the computation at point ***(i,j)*** depends on data at adjacent (or "nearby") points. Adjacent dependencies are often represented graphically using a “stencil” as shown in

Figure 2-12 and Figure 2-13.

```
x(i,j) = y(i,j) + y(i+1,j) + y(i-1,j) + y(i,j-1) + y(i,j+1)
```

“Stencil”: `x(i,j)` **depends on**

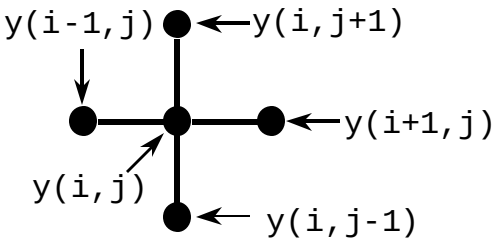


Figure 2-12 This five point stencil illustrates the dependencies of the array *y* on the computation of x.

```
x(i,j) = y(i,j) + y(i+1,j) + y(i+2,j) + y(i,j+1) + y(i,j+2)
                + y(i-1,j) + y(i-2,j) + y(i,j-1) + y(i,j-2)
                + y(i+1,j+1) + y(i+1,j-1)
                + y(i-1,j+1) + y(i-1,j-1)
```

"Stencil": `x(i,j)` **depends on**

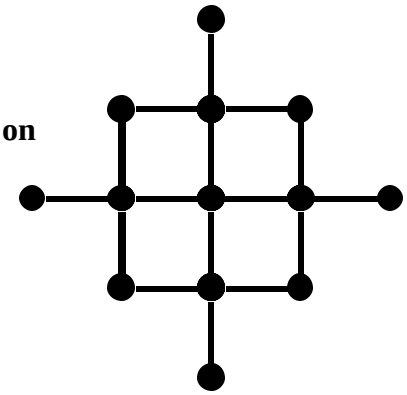


Figure 2-13: A thirteen point stencil illustrates the dependencies required when *x* must access data two points in each direction on *y* in the code segment shown.

In Figure 2-14 stencils have been overlaid on graphical representations of the sub-domains assigned to each process during a run made on three processes. The stencil centered at global point *(2,2)* on process P1 illustrates that computations at this grid point require values from global points *(2,2)*, *(2,1)*, *(1,2)*, *(2,3)*, and *(3,2)*. These array elements are all inside the local sub-domain of process P1. Similarly, computations at global point *(5,8)* depend only on array elements inside the local sub-domain of process P2. However, computations on sub-domain boundaries cannot be performed so easily. For example, the stencil centered at global point *(7,5)* on process P2 depends on the element at global point *(8,5)* which is located in the local sub-domain of process P3. Similarly, the stencil centered at global point *(8,2)* on process P3 requires an element from process P2. The results of the parallel program above are incorrect because no data is sent between processes to resolve the adjacent dependence in loop 200.

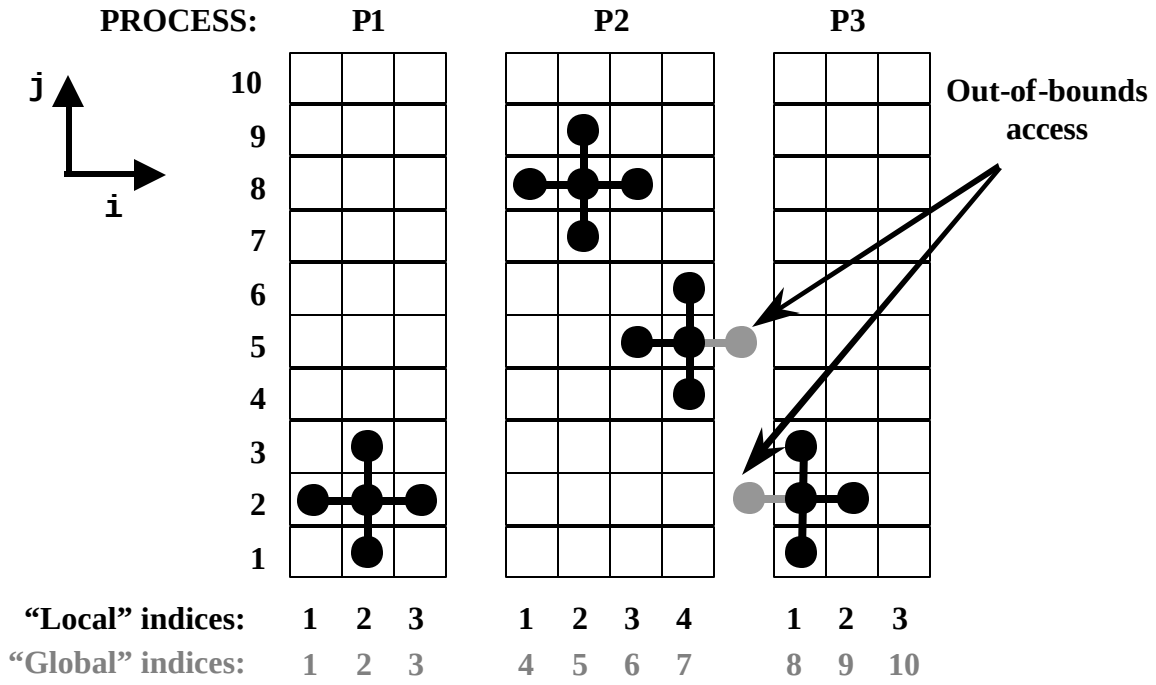


Figure 2-14: Illustration of how an adjacent dependency causes out of bounds data references on processes P2 and P3.

It is possible to solve this problem by sending single data points between processes. However, on high-latency machines, sending messages that contain only one array element is very inefficient compared to sending messages that contain many array elements. The most common approach to handle adjacent dependencies is to create "halo regions" to store these data as shown in Figure 2-15. When data in these regions are needed, the halo regions are updated by swapping columns (or larger blocks) of data between processes as shown in Figure 2-16. This form of inter-process communication is called "exchange" and is supported by the EXCHANGE directive.

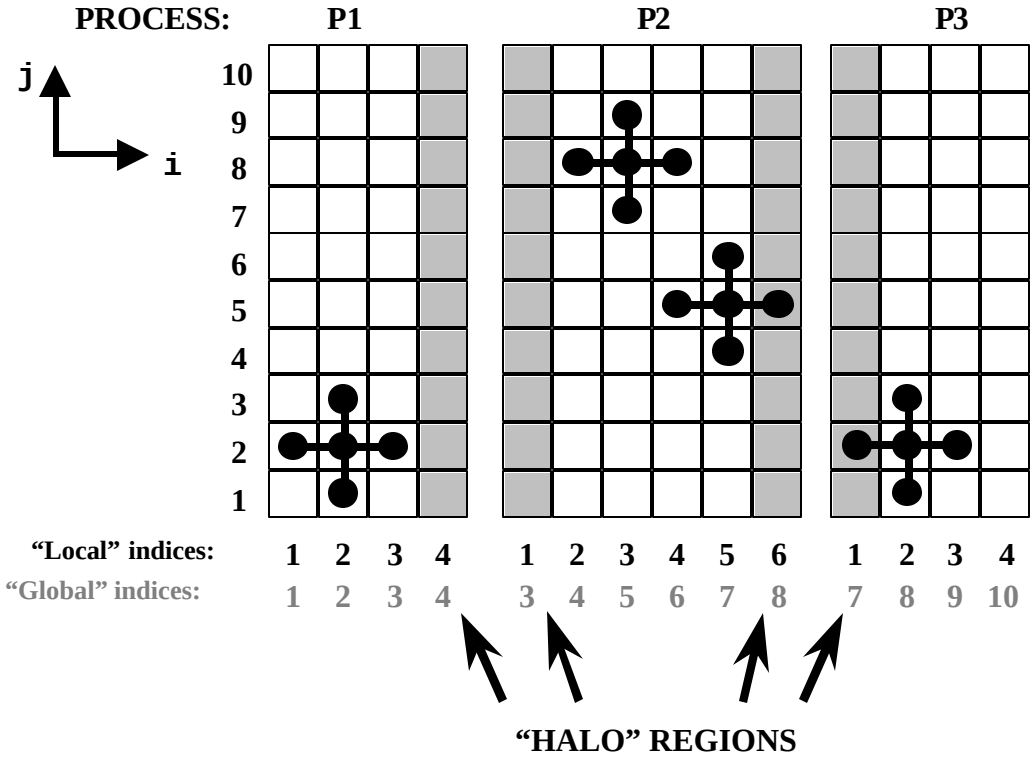


Figure 2-15: Halo regions eliminate the out of bounds array references.

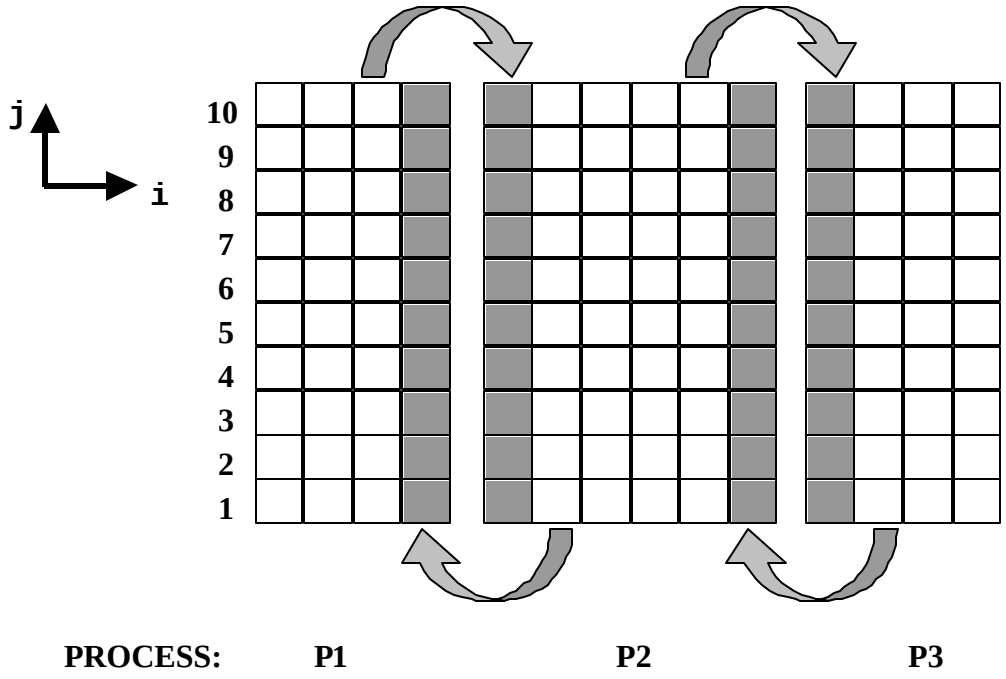


Figure 2-16: Halo regions are updated by exchanging data between adjacent processes.

Below is a corrected parallel program that uses halo regions and includes exchange communication:

[Source file: laplace.f]

```
 1        program laplace
 2        include 'basic.inc'
 3        im = 10
 4        jm = 10
 5  CSMS$CREATE_DECOMP(DECOMP_I, <im>, <1>)
 6        call laplace
 7        end
 8
 9        subroutine laplace
10        include 'basic.inc'
11        integer i, j, iter
12        real max_error
13        real tolerance
14        parameter (tolerance = 0.001)
15  CSMS$DISTRIBUTE(DECOMP_I, <im>) BEGIN
16        real f(im,jm), df(im,jm)
17  CSMS$DISTRIBUTE END
18  CSMS$PARALLEL(DECOMP_I,<i>) BEGIN
19        do 100 j=1,jm
20        do 100 i=1,im
21          f(i,j) = 0.0
22    100 continue
23        do 110 j=1,jm
24  CSMS$GLOBAL_INDEX(1) BEGIN
25          f( 1,j) = 2.0
26          f(im,j) = 2.0
27  CSMS$GLOBAL_INDEX END
28    110 continue
29        do 120 i=1,im
30          f(i, 1) = 2.0
31          f(i,jm) = 2.0
32    120 continue
33        iter = 0
34        max_error = 2.0 * tolerance
35  C main iteration loop...
36        do while ((max_error .gt. tolerance) .and. (iter .lt. 1000))
37          iter = iter + 1
38          max_error = 0.0
39  CSMS$EXCHANGE(f)
40          do 200 j=2,jm-1
41          do 200 i=2,im-1
42            df(i,j) = 0.25*(f(i-1,j) + f(i+1,j) + f(i,j-1) + f(i,j+1))
43       &                - f(i,j)
44    200 continue
```

```
45          do 300 j=2,jm-1
46          do 300 i=2,im-1
47            if (max_error .lt. abs(df(i,j))) then
48              max_error = abs(df(i,j))
49            endif
50    300 continue
51  CSMS$REDUCE(max_error, MAX)
52          do 400 j=2,jm-1
53          do 400 i=2,im-1
54            f(i,j) = f(i,j) + df(i,j)
55    400 continue
56        enddo
57  CSMS$PARALLEL END
58        print *, 'Solution required ',iter,' iterations'
59        print *, 'Final error = ', max_error
60
61        return
62        end
```

Example 2-5: The laplace program which has been corrected to exchange the array *f*. Resolves the adjacent dependencies in loop 200.

The third parameter of CREATE_DECOMP directive has been changed to ***<1>***. This indicates that all arrays decomposed using ***DECOMP_I*** will have a halo region one point thick added in the first decomposed dimension (the ***i*** dimension in this case). The EXCHANGE directive has been added on line 39. Its only parameter is the name of the variable (***f)*** to be exchanged. The EXCHANGE directive is placed immediately before loop 200 to ensure that halo regions of ***f*** are updated prior to the computations that need them. The EXCHANGE directive is described in more detail in section 5.1.

Now the parallel program produces the correct results on more than one process:

```
>> smsRun 2 laplace_sms
 Solution required  85  iterations
 Final error =  9.9968910E-4

>> smsRun 3 laplace_sms
 Solution required  85  iterations
 Final error =  9.9968910E-4
```

Notice that only interior process P2 has halo regions on both sides in Figure 2-15. A current limitation of SMS is that it only supports non-periodic boundary conditions. Therefore, halo regions are only needed on one side of processes that are on the edge of a global array (i.e. processes P1 and P3). This limitation will be removed in a future SMS release.

2.6 Writing Output to Disk

The Laplace solver (Example 2-5) would be more useful if the final state of array *f* could be written to disk. This is easily done by adding the following code fragment immediately before the *return* statement (line 61) in subroutine *laplace*:

```
      open(10, file='f.out', form='unformatted')
      write(10) f
      close(10)
```

When the serial program is run, file *f.out* is written. For the SMS parallel program, no additional directives are required to handle this output. By default, SMS automatically generates *f.out* in exactly the same format as the serial program, for any number of processes. However, SMS can also produce other file formats as discussed in Section 9.

2.7 Using Multiple Decompositions

So far, we have seen how to parallelize a program that only requires a single domain decomposition. However, many programs require the use of different decompositions at different times to run efficiently in parallel. The TRANSFER directive provides the means to transform arrays between decompositions. Spectral NWP models are a prime candidates for application of TRANSFER (see Section 6).

In this section, we present a simple case where two different decompositions are needed. In Example 2-6, the statement at line 42 contains a dependency called a "recurrence relation". In this statement, an update to x(i,j) depends on x(i,j-1) which was updated in the previous loop iteration. SMS does not currently provide directives that directly support parallelization of this type if the array dimension is decomposed. SMS will support simple one-dimensional recurrence relations in a future release. In this example, the second (*j*) dimension is decomposed, so SMS cannot handle this statement. Similarly, the loop starting at line 61 prevents decomposition in *i*. One solution, given in Example 2-6, is to decompose *x* in *i* and *y* in *j*.

```
[transfer.inc]
 1        integer im, jm
 2        common /sizes_com/ im, jm
 3
 4  CSMS$DECLARE_DECOMP(DECOMP_I)
 5  CSMS$DECLARE_DECOMP(DECOMP_J)
 6

[transfer.f]
 1        program TRANSFER1
 2        implicit none
 3
 4        include 'transfer.inc'
 5
 6        integer i
```

```
 7       integer j
 8
 9       im = 60
10       jm = 90
11
12  CSMS$CREATE_DECOMP(DECOMP_I, <im>, <0>)
13  CSMS$CREATE_DECOMP(DECOMP_J, <jm>, <0>)
14
15       call DO_IT
16
17       end
18
19
20
21       subroutine DO_IT
22       include 'transfer.inc'
23
24  CSMS$DISTRIBUTE(DECOMP_I, im) BEGIN
25       real x(im,jm)
26  CSMS$DISTRIBUTE END
27
28  CSMS$DISTRIBUTE(DECOMP_J, jm) BEGIN
29       real y(im,jm)
30  CSMS$DISTRIBUTE END
31
32  C BEGIN
33
34       x = 1.0
35
36  CSMS$PARALLEL(DECOMP_I, <i>) BEGIN
37
38  C dependency in the j dimension that
39  C SMS does not provide directives to parallelize
40       do j = 2, jm
41         do i = 1, im
42           x(i,j) = x(i,j) + x(i,j-1)
43         end do
44       end do
45  CSMS$PARALLEL END
46
47  CSMS$TRANSFER(<X, Y>) BEGIN
48       do j = 1, jm
49         do i = 1,im
50           y(i,j) = x(i,j)
51         end do
52       end do
53  CSMS$TRANSFER END
54
55       call CALCS_THAT_MODIFY_X(x)
56
57  CSMS$PARALLEL(DECOMP_J, <j>) BEGIN
58
59  C dependency in the i dimension that
60  C SMS does not provide directives to parallelize
```

```
61        do j = 1, jm
62          do i = 2, im
63            y(i,j) = y(i,j) + y(i-1,j)
64          end do
65        end do
66  CSMS$PARALLEL END
67
68        open(10,file='f1',form='unformatted')
69        write(10) y
70        close(10)
71
72        return
73        end
```

Example 2-6: A simple SMS parallel program that requires two data decompositions due to recurrance relations in "x" and "y".

Example 2-6 contains two DECLARE_DECOMP and CREATE_DECOMP directives. The DISTRIBUTE directive at line 24 uses ***DECOMP_I*** to decompose *x* in *i*. The DISTRIBUTE directive at line 28 uses ***DECOMP_J*** to decompose *y* in *j*. The TRANSFER directive at line 47 generates the communication to transpose *x* into *y* as illustrated in Figure 2-17. SMS implements this by replacing the code between the BEGIN and END TRANSFER directives with a call to a subroutine that does the transposition. *x* is referred to as the source array of the TRANSFER directive and *y* is referred to as the destination array. The type and rank of the source and destination arrays must be the same. However, the array sizes may differ.

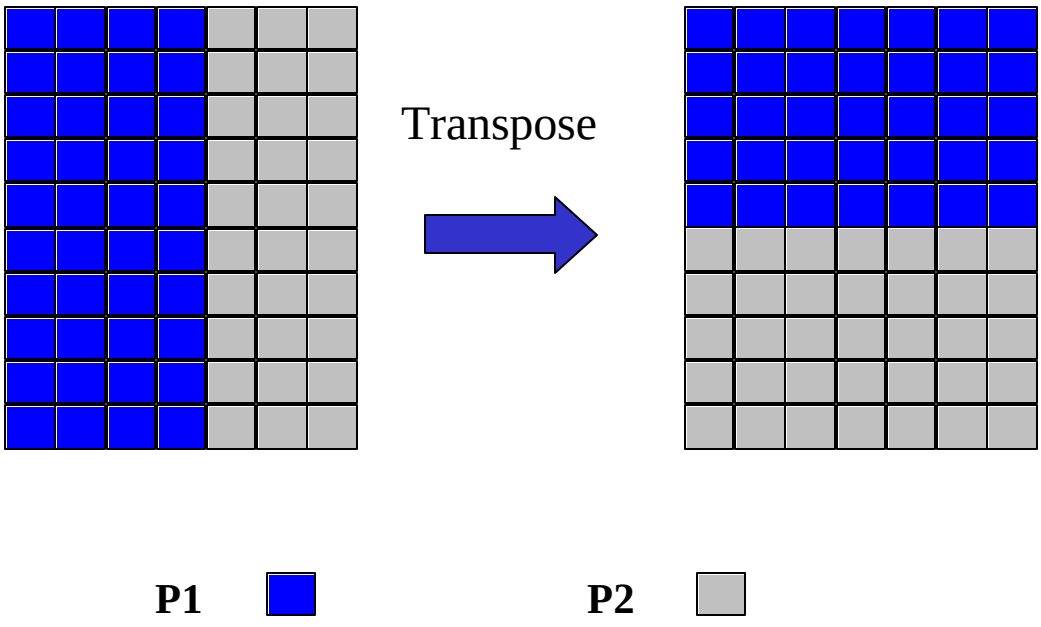


Figure 2-17. An illustration of the data movement required between processes P1 and P2 for a transposition operation.

3 Decomposing Arrays and Parallelizing Loops

3.1 Choosing Decompositions

In order to choose domain decompositions that will allow optimal performance, the dependencies of arrays on one another must be analyzed. Usually, several decomposition options are possible. Decompositions of 3D arrays supported by SMS are shown in Figure 3-1. The dependence analysis is used to help pick optimal decompositions that will minimize inter-process communication. Typical FDA NWP models will be optimally decomposed in one or both of the horizontal dimensions as illustrated "a", "b", or "d" of Figure 3-1. Decompositions used by typical spectral NWP models are described in Section 6.2.

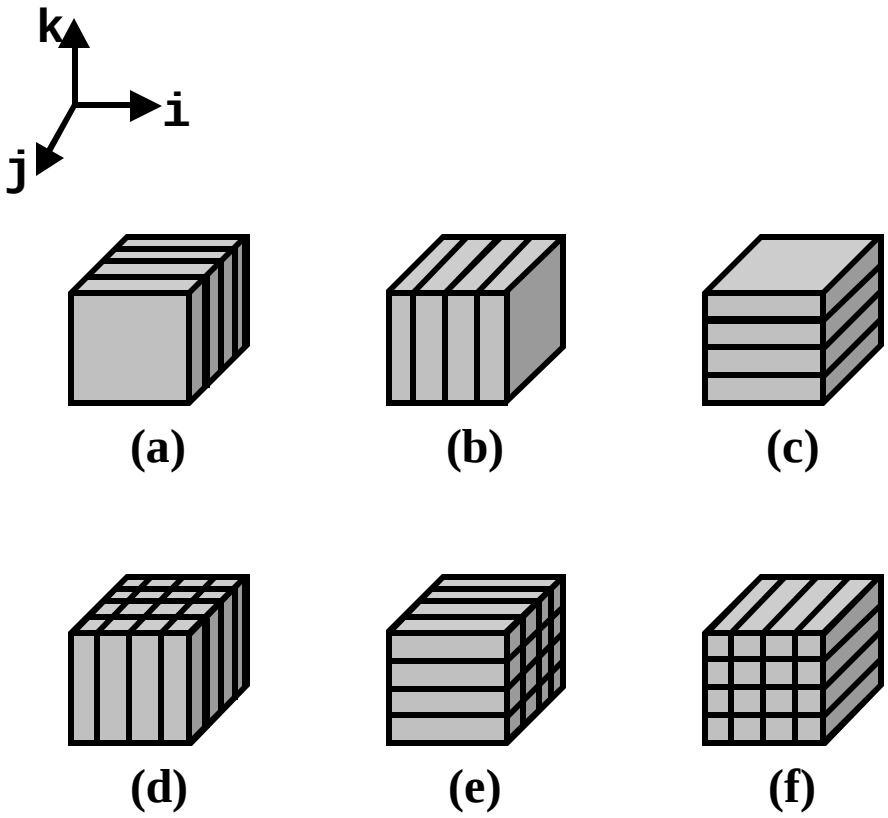


Figure 3-1: Three-dimensional decompositions supported by SMS.

Other issues to consider when selecting decompositions are the architecture of the machine on which the program will most likely be run and how many processes will be available. For vector machines, it is best to leave the inner dimension non-decomposed when possible to maximize vector lengths. On cache-based machines, it may be best to decompose the inner dimension instead. For example, in Figure 3-1, decomposition "a" would preserve long vector lengths while decomposition "b" would not. If the number of processes available were larger than the number

of grid points in the single decomposed dimension, two dimensions would have to be decomposed.

3.2 Two-Dimensional Decompositions

The full power of the DECLARE_DECOMP, CREATE_DECOMP, DISTRIBUTE, and PARALLEL directives becomes more apparent when two dimensions are decomposed. Consider the following example:

```
[Include file:  decomp_ex1.inc]

 1        integer im, jm, km
 2        common /sizes_com/ im, jm, km
 3  CSMS$DECLARE_DECOMP(DECOMP_IJ)


[Source file:  decomp_ex1.f]

 1        program decomp_ex1
 2        include 'decomp_ex1.inc'
 3        im = 15
 4        jm = 10
 5        km = 2
 6  CSMS$CREATE_DECOMP(DECOMP_IJ, <im, jm>, <0,0>)
 7        call compute
 8        end
 9
10        subroutine compute
11        include 'decomp_ex1.inc'
12        integer i, j, k
13  CSMS$DISTRIBUTE(DECOMP_IJ, <im>, <jm>) BEGIN
14        integer z(im,jm,km)
15  CSMS$DISTRIBUTE END
16        integer zsum
17  CSMS$PARALLEL(DECOMP_IJ,<i>,<j>) BEGIN
18        do 100 k=1,km
19        do 100 j=1,jm
20        do 100 i=1,im
21          z(i,j,k) = 1
22    100 continue
23        zsum = 0
24        do 200 k=1,km
25        do 200 j=1,jm
26        do 200 i=1,im
27          zsum = zsum + z(i,j,k)
28    200 continue
29  CSMS$PARALLEL END
30  CSMS$REDUCE(zsum, SUM)
31        print *,'zsum = ',zsum
32        return
33        end
```

Example 3-1: An SMS program that uses a two dimensional decomposition.

When run, the serial version of this program prints the following message:

```
>> decomp_ex1
 zsum =          300
```

Directives CREATE_DECOMP, DISTRIBUTE, and PARALLEL now have more complex parameters than in the simple examples from Section 2.3. The second parameter to CREATE_DECOMP, ***<im, jm>***, indicates that the decomposition named ***DECOMP_IJ*** has two decomposed dimensions and that the global size of the first decomposed dimension is ***im*** and the global size of the second decomposed dimension is ***jm***. The third parameter, ***<0,0>***, indicates that ***DECOMP_IJ*** has no halo regions in either decomposed dimension.

The second parameter to DISTRIBUTE, ***<im>***, indicates that array dimensions of size ***im*** are decomposed as described by the first decomposed dimension of ***DECOMP_IJ***. The third parameter, ***<jm>***, indicates that array dimensions of size ***jm*** are decomposed as described by the second decomposed dimension of ***DECOMP_IJ***. So, the first dimension of array ***z*** is decomposed as described by the first decomposed dimension of ***DECOMP_IJ*** and the second dimension of array ***z*** is decomposed as described by the second decomposed dimension of ***DECOMP_IJ***. The third dimension of array ***z*** will not be decomposed. This is decomposition "d" in Figure 3-1. More details about DISTRIBUTE can be found in Section 3.5.

The second parameter to PARALLEL, ***<i>***, is used to identify loop indices for loops spanning the first decomposed dimension of ***DECOMP_IJ***. Similarly, the third parameter, ***<j>,*** is used to identify loop indices for loops spanning the second decomposed dimension of ***DECOMP_IJ***. The PARALLEL directive will translate both the ***i*** and ***j*** dimensions of loops 100 and 200 to local loop bounds.

When this code is run on 2 or 3 processes, we see the expected results:

```
>> smsRun 2 decomp_ex1_sms
 zsum =  300
>> smsRun 3 decomp_ex1_sms
 zsum =  300
```

3.3 Decomposing Arrays that use Statically Allocated Memory

When dynamic memory allocation is used, SMS automatically sets local array sizes at run-time. In contrast, when static memory allocation is used, local array sizes must be set by the programmer. Therefore, it is essential to understand how SMS will assign processes to decomposed dimensions to avoid slowing execution down on cache machines and wasting memory on any machine. Even when dynamic memory allocation is used it is useful to understand process assignment when tuning performance.

3.3.1 How SMS Assigns Processes to Decomposed Dimensions

To better understand process assignment, subroutine "compute" has been modified to print out the number of array elements each process has in each dimension. The following code replaces the print statement on line 31 of Example 3-1:

```
CSMS$PRINT_MODE(ORDERED) BEGIN
      print *,'  MY  im = ',i-1,'  jm = ',j-1,'  km = ',k-1
CSMS$PRINT_MODE END
```

The "ORDERED" print mode ensures that each process prints a message and that messages always appear in the same order. The ORDERED print mode only works when all processes execute the enclosed print statement(s). Print modes are discussed in detail in Section 9.

Assume the new program is named *decomp_ex2_sms*. When it is run on one process, the following results are printed on the screen:

```
>> smsRun 1 decomp_ex2_sms
   MY  im =  15   jm =  10   km =  2
```

The results of the one-process run for a single *k* plane of array *z* indicate that loops spanned the full array dimensions *(15,10,2)*, as shown in Figure 3-2.

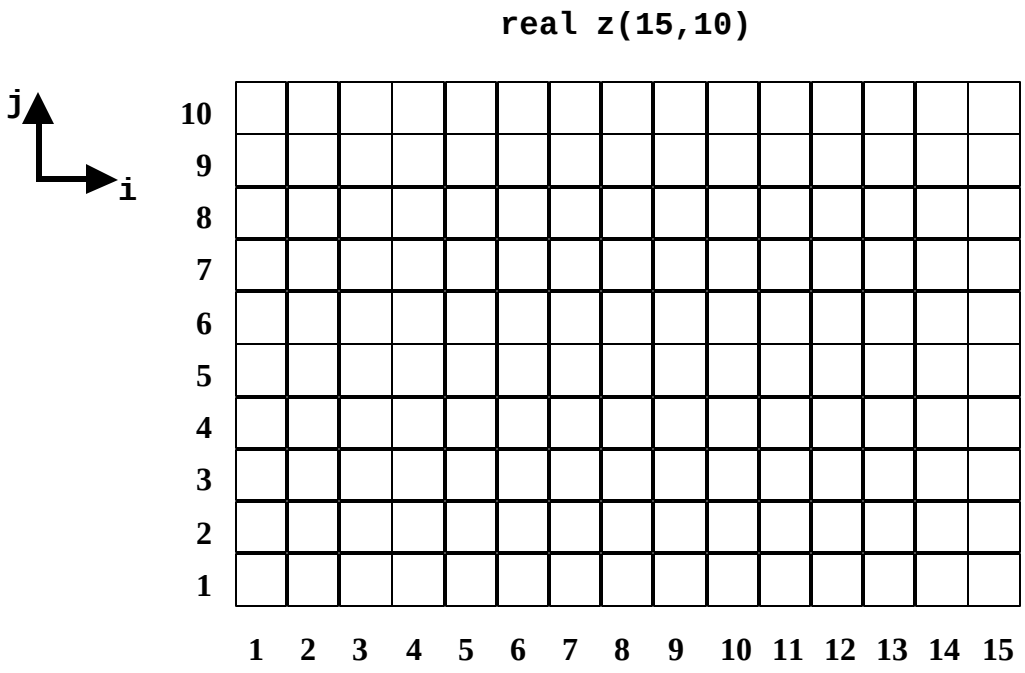


Figure 3-2: Illustration of one "k" plane of the array *z* required to support a one process run. In this example, loops will span the entire array.

On two processes:

```
>> smsRun 2 decomp_ex2_sms
   MY  im =  8   jm =  10   km =  2
   MY  im =  7   jm =  10   km =  2
```

Here, one process's loops spanned *(8,10,2)* and the second process's loops spanned *(7,10,2)*. In the two process run, SMS decomposed the array in the first dimension as illustrated in Figure 3-3.

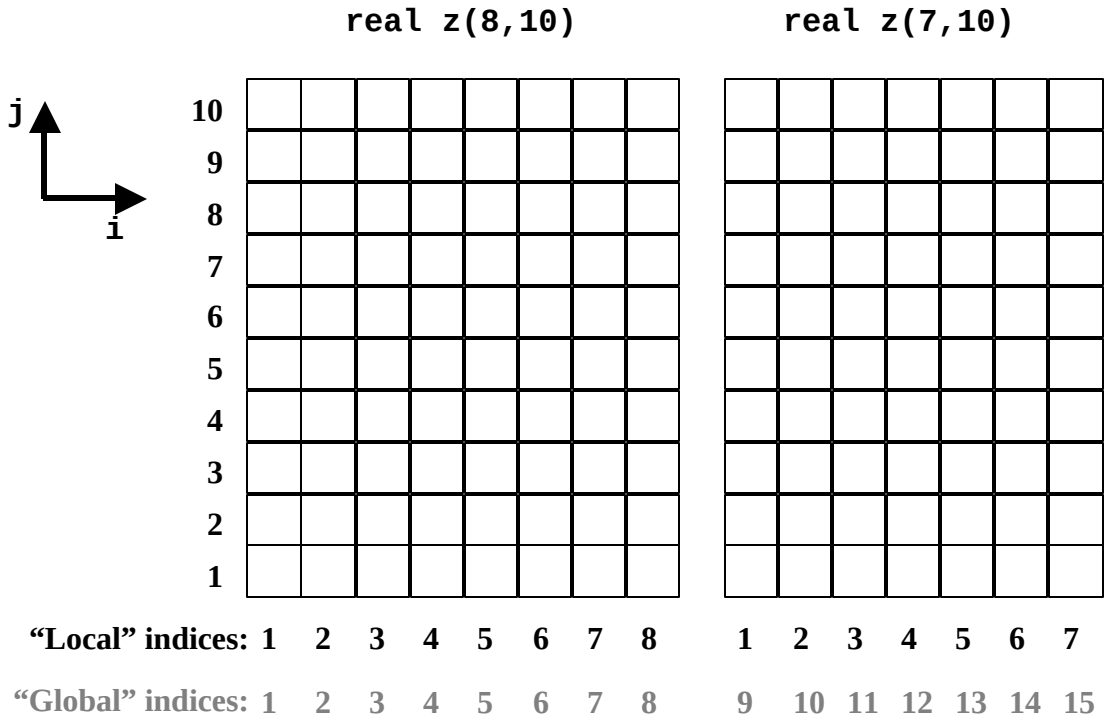


Figure 3-3. For a two process run, SMS assigns two processes to the first decomposed dimension (*im*) and leaves the second decomposed dimension non-decomposed.

On three processes:

```
>> smsRun 3 decomp_ex2_sms
   MY  im =  5   jm =  10   km =  2
   MY  im =  5   jm =  10   km =  2
   MY  im =  5   jm =  10   km =  2
```

In this case, each of the processes' loops spanned *(5,10,2)*. SMS assigned three processes to the first decomposed dimension (*im*) and left the second decomposed dimension non-decomposed. On 4 processes:

```
>> smsRun 4 decomp_ex2_sms
   MY  im =  8   jm =  5   km =  2
   MY  im =  7   jm =  5   km =  2
   MY  im =  8   jm =  5   km =  2
   MY  im =  7   jm =  5   km =  2
```

Here, two of the process's loops spanned *(8,5,2)* and the other two process's loops spanned *(7,5,2)*. SMS assigned two "columns" of processes to the first decomposed dimension (*im*) and two "rows" of processes to the second decomposed dimension (*jm*) as shown in Figure 3-4.

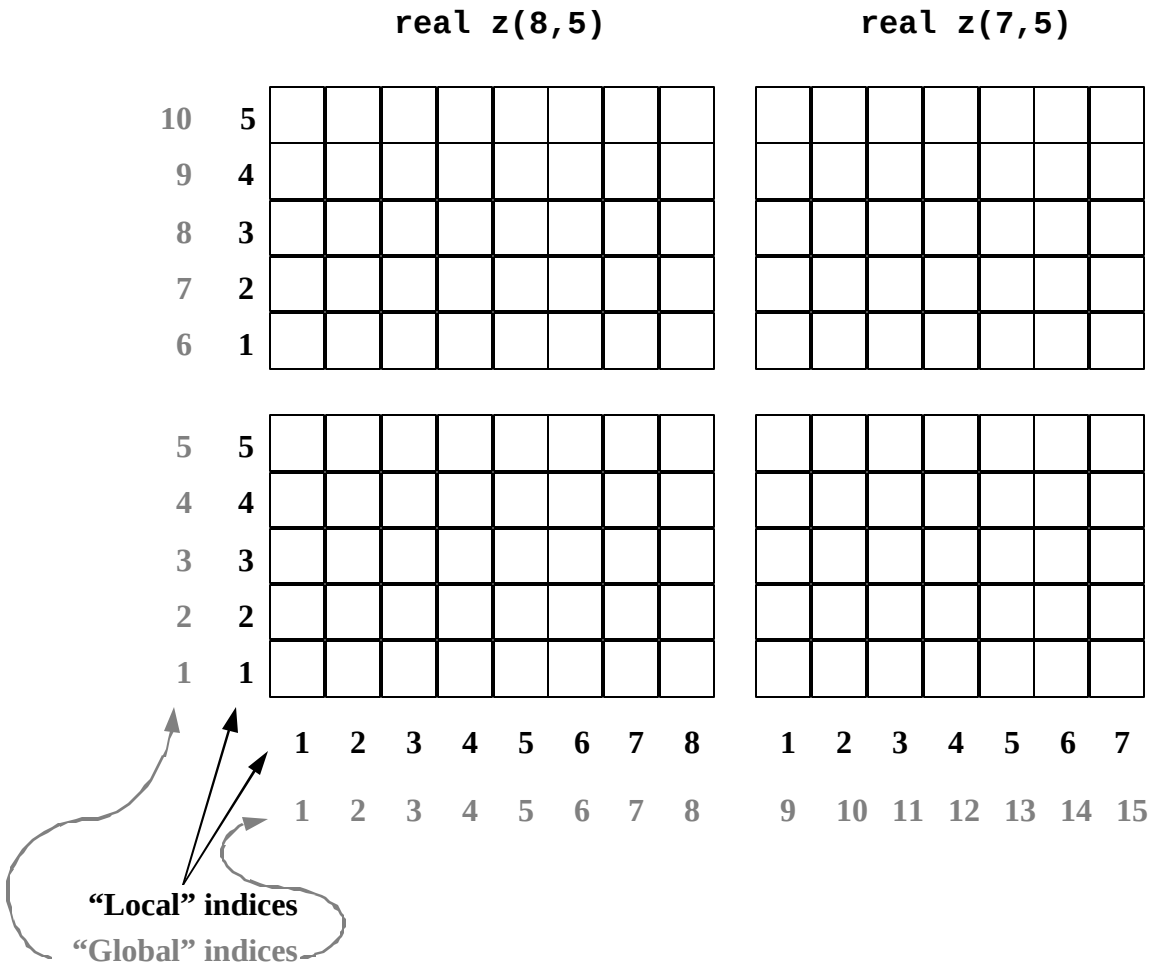


Figure 3-4: For a four process run, SMS decomposes in both dimensions.

On eight processes:

```
>> smsRun 8 decomp_ex2_sms
   MY  im =  4   jm =  5   km =  2
   MY  im =  4   jm =  5   km =  2
   MY  im =  4   jm =  5   km =  2
   MY  im =  3   jm =  5   km =  2
   MY  im =  4   jm =  5   km =  2
   MY  im =  4   jm =  5   km =  2
   MY  im =  4   jm =  5   km =  2
   MY  im =  3   jm =  5   km =  2
```

In this case, six of the process's loops spanned *(4,5,2)* and two of the process's loops spanned *(3,5,2)*. Here, SMS has assigned four "columns" of processes to the first decomposed dimension (*im*) and two "rows" of processes to the second decomposed dimension (*jm*). This is illustrated in Figure 3-5.

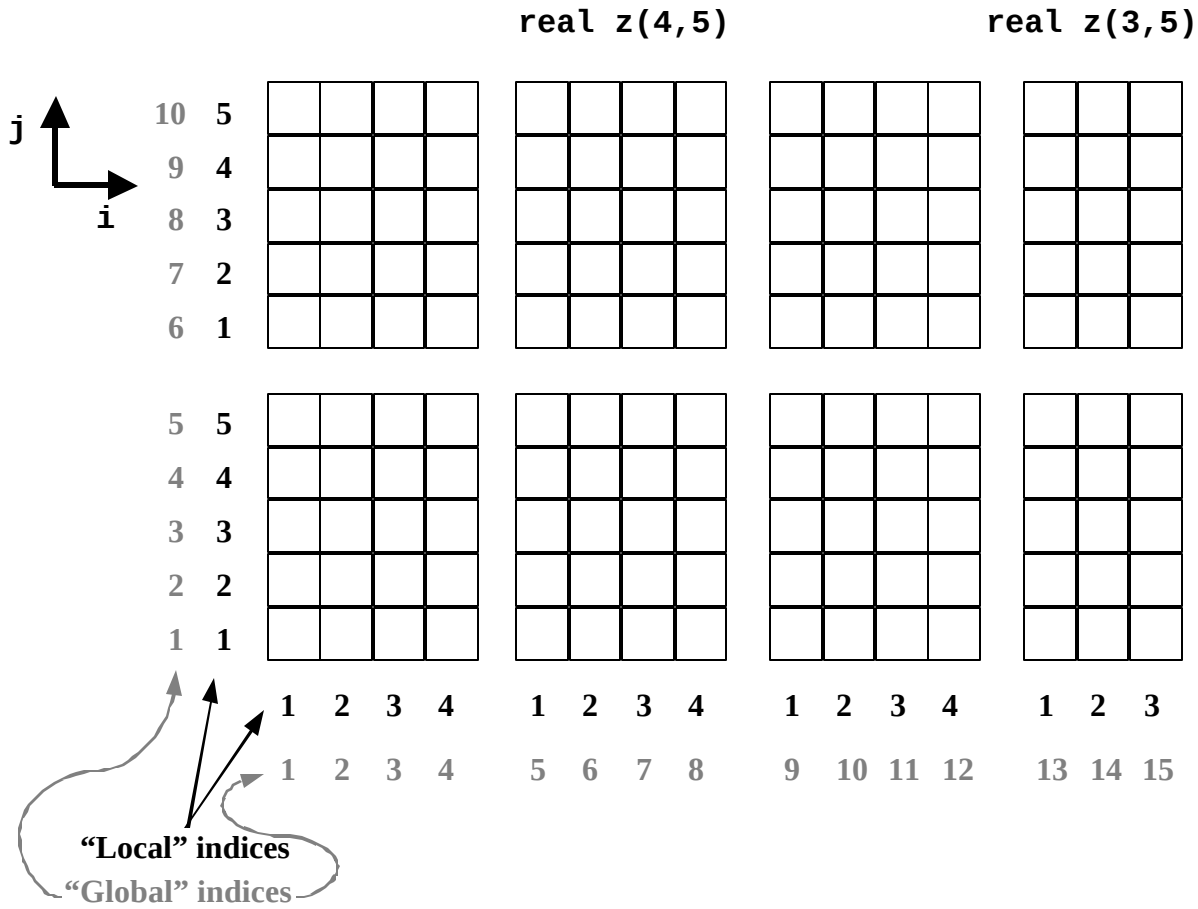


Figure 3-5: In the eight process run, SMS assigns four processes to the first decomposed dimension and two to the second.

From these results, it can be seen that SMS will assign more processes to the decomposed dimension with the largest global size, when possible. When global sizes of decomposed dimensions are equal, SMS will assign more processes to the second decomposed dimension. Also, SMS will always attempt to make process layout as close to "square" as possible. The rules followed by SMS to assign processes to decomposed dimensions are described in detail in Appendix A. However, it may be easier to simply print local sizes as in the previous example. A future SMS release will ease the process of setting local array sizes in the static case and will print out the process layout for each decomposition when it is created.

3.3.2 A Static Memory Program

Example 3-2 illustrates a program using static memory allocation. In this example, the DECLARE_DECOMP directive requires a new second parameter, ***<(im/2)+1, jm/2>.*** This informs the translator that the decomposition named ***DECOMP_IJ*** has two decomposed

dimensions. It also indicates that ***DECOMP_IJ*** will be used for arrays that are statically allocated and that the DISTRIBUTE command should translate sizes of declared array dimensions corresponding to the first and second decomposed dimensions to local sizes ***(im/2)+1*** and ***jm/2*** respectively.

```
[Include file:  decomp_ex4.inc]

 1       integer im, jm, km
 2       parameter (im = 15, jm = 10, km = 2)
 3 CSMS$DECLARE_DECOMP(DECOMP_IJ, <(im/2)+1, jm/2>)

[Source file:  decomp_ex4.f]

 4       program decomp_ex4
 5       include 'decomp_ex4.inc'
 6 CSMS$DISTRIBUTE(DECOMP_IJ, <im>, <jm>) BEGIN
 7       integer z(im,jm,km)
 8 CSMS$DISTRIBUTE END
 9       integer zsum, i, j, k
10 CSMS$CREATE_DECOMP(DECOMP_IJ, <im, jm>, <0,0>)
11 CSMS$PARALLEL(DECOMP_IJ,<i>,<j>) BEGIN
12       do 100 k=1,km
13      do 100 j=1,jm
14       do 100 i=1,im
15         z(i,j,k) = 1
16   100 continue
17       zsum = 0
18       do 200 k=1,km
19       do 200 j=1,jm
20       do 200 i=1,im
21         zsum = zsum + z(i,j,k)
22   200 continue
23 CSMS$PARALLEL END
24 CSMS$REDUCE(zsum, SUM)
25       print *,'zsum = ',zsum
26       end
```

Example 3-2: An SMS program that uses static memory allocation requires the local sizes be declared in the DECLARE_DECOMP directive. In this example, these local sizes are: *(im/2)+1* and *jm/2*.

In static memory cases such as this where the number of processes assigned to a decomposed dimension does not evenly divide the global size of that dimension, the local sizes used in the DECLARE_DECOMP directive must be set for the process(es) that use(s) the most memory. As we saw in Figure 3-4, these are precisely the local sizes needed by SMS for a four-process run. The term ***(im/2)+1*** takes into account the fact that two of the processes requires local arrays of size *(8,5,2)* while the other two requires arrays of size *(7,5,2)* as illustrated in Figure 3-6.

Since arrays are declared statically, the rules of Fortran77 require that ***(im/2)+1*** and ***jm/2*** be compile-time constants in order to be used in a declaration statement. The translator handles this by generating appropriate parameter statements during the translation of the

DECLARE_DECOMP directive. These parameter statements are then used during translation of array sizes inside DISTRIBUTE directives. Conceptually, the declaration of *z* on line 7 of Example 3-2 will be translated to:

```
integer z((im/2)+1,jm/2,km)
```

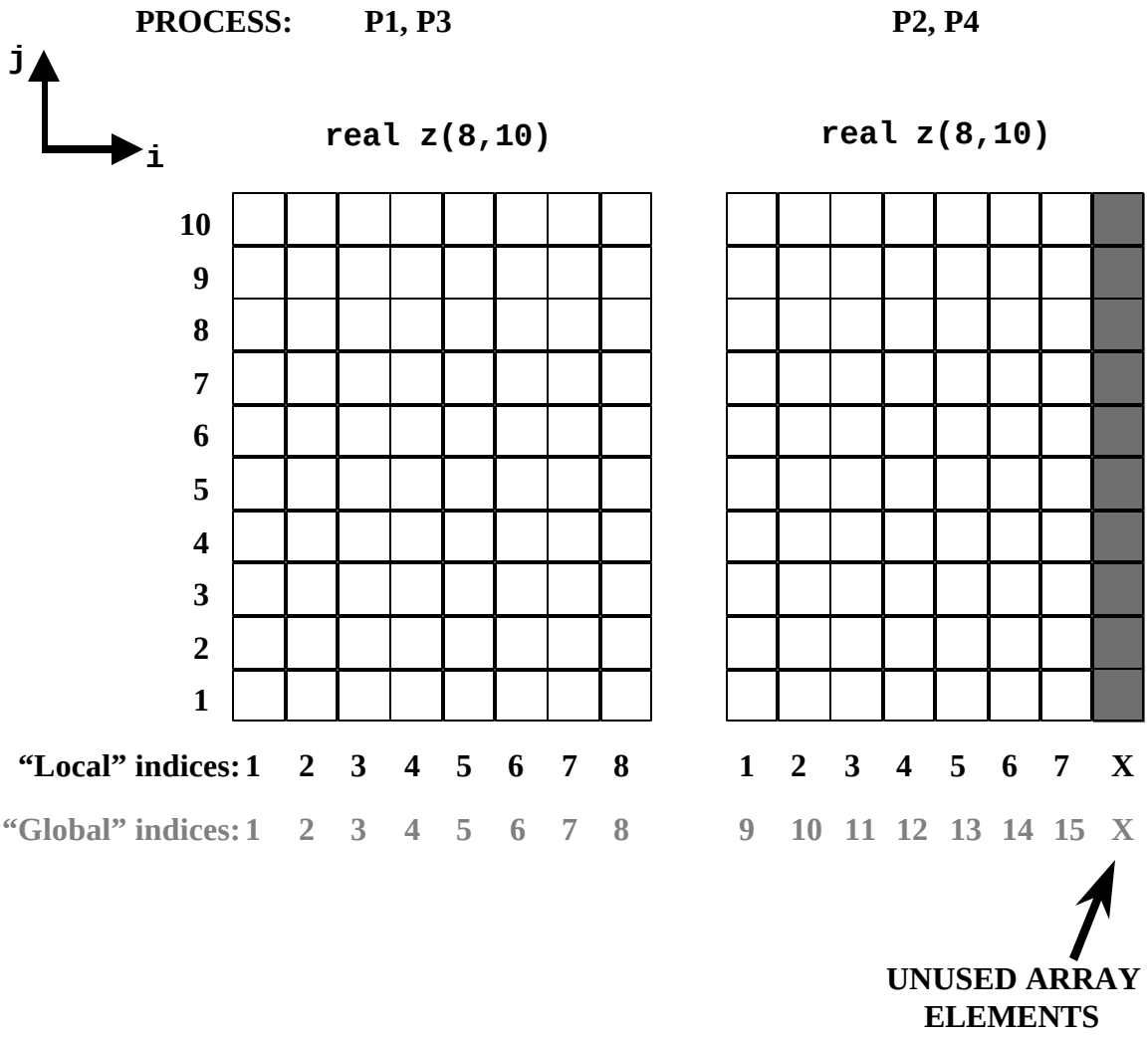


Figure 3-6: For static memory allocation, the size of the decomposed arrays is set in the DECLARE_DECOMP directive based on the number of processes that will be used to run the program. Sometimes all the memory declared will not be used as illustrated in processes P2 and P4.

A run made on 8 processes yields expected results. However, a run made on 2 processes produces the following:

```
>> smsRun 2 decomp_ex4_sms
 Process:   1 Error at: ./decomp_ex4_sms.f.tmp:10.1
 Process:   1 Error status=   -2202 MSG: DECOMPOSED ARRAYS ARE TOO SMALL.
 Process:   1 Aborting...
```

What happened? From Example 3-2, we saw that the largest local array sizes required on any process for the two-process run is *(8,10,2)*. However, the DECLARE_DECOMP directive set local array sizes to ***((im/2)+1,jm/2,km)*** = *(8,5,2)* which is too small for the two process run (see Figure 3-6). SMS detects this error at run time, prints the error messages, and aborts the program.

Why did it work for the 8-process run? Again, from Example 3-2, we saw that the largest local array sizes required on any process for the eight-process run are *(4,5,2)*. So the local array sizes were big enough to hold the translated arrays and the program ran as expected. However, it wasted memory because only half of each declared array was ever used *(1:4,*,*)*.

In addition to wasting memory, performance of the 8-process run might not be optimal on a cache-based machine because the data used in each array are scattered over a block of memory twice the needed size. This is likely to result in more cache misses and may degrade performance, sometimes significantly. This effect becomes more severe as the number of processes increases. For example, if the program were run on 32 processes, the largest local array sizes required on any process would be only *(2,3,2)*. Therefore, it is especially important to declare arrays using the smallest possible sizes for large numbers of processes.

To fix the local arrays sizes for a two-process run, we can modify the sizes in the DECLARE_DECOMP directive as follows:

```
CSMS$DECLARE_DECOMP(DECOMP_IJ, <(im/2)+1, jm>)
```

If the following DECLARE_DECOMP directive were used

```
CSMS$DECLARE_DECOMP(DECOMP_IJ, <im, jm>)
```

all translated arrays would be declared full-size. This code could then be run on any number of processes (provided each process has enough memory). This is very useful during debugging because one common technique for finding bugs in a parallel code is to compare results for runs made on different numbers of processes. Once debugging is complete, the DECLARE_DECOMP directives should be changed to minimize memory use.

In summary, SMS provides the flexibility of allowing memory to be wasted for convenience during debugging. However, the user should try to minimize memory waste once debugging is complete. Failure to conserve memory can result in performance degradation on cache-based machines.

3.4 More About DECLARE_DECOMP and CREATE_DECOMP

3.4.1 Placement of DECLARE_DECOMP and CREATE_DECOMP

It is important to understand a few details concerning DECLARE_DECOMP and CREATE_DECOMP so these directives can be placed correctly. The SMS code translator, PPP, converts a DECLARE_DECOMP directive into Fortran declarations of all the variables needed to store the internal description of an SMS decomposition. So, a DECLARE_DECOMP directive must be placed before the first executable line of code in a program. Also, if a decomposition needs to be visible to more than one program unit, then it is best to place the DECLARE_DECOMP directive in an include file. A CREATE_DECOMP directive is translated into executable Fortran code that initializes all the internal variables declared in the translation of the corresponding DECLARE_DECOMP directive. A CREATE_DECOMP statement may only be placed where it would be legal to write an executable line of Fortran code.

The rules for placing the CREATE_DECOMP and DISTRIBUTE directives differ for programs that use static or dynamic memory. The CREATE_DECOMP directive can actually appear after a DISTRIBUTE directive in the static memory case. However, in the dynamic memory case this is not possible because number of decomposed dimensions is not known until the CREATE_DECOMP directive is reached. In this case, the code generated by CREATE_DECOMP must execute prior to any subroutine containing DISTRIBUTE directives.

3.4.2 Load Balancing via Index Scrambling

Ideally, each process will have exactly the same amount of work to do. In practice, most NWP models have computations that vary spatially so some processes may have more work to do than others. This is commonly known as load imbalance. Load imbalances slow down a parallel program because some processes with less work are forced to wait for processes with more work to catch up. One example is load imbalance in a global NWP model due to differences in computation required for day and night grid-points. In this case more computation is required at longitudes where the sun shines. There are also load imbalances between latitudes in the northern and southern hemispheres during winter or summer. Figure 3-7 illustrates longitude scrambling.

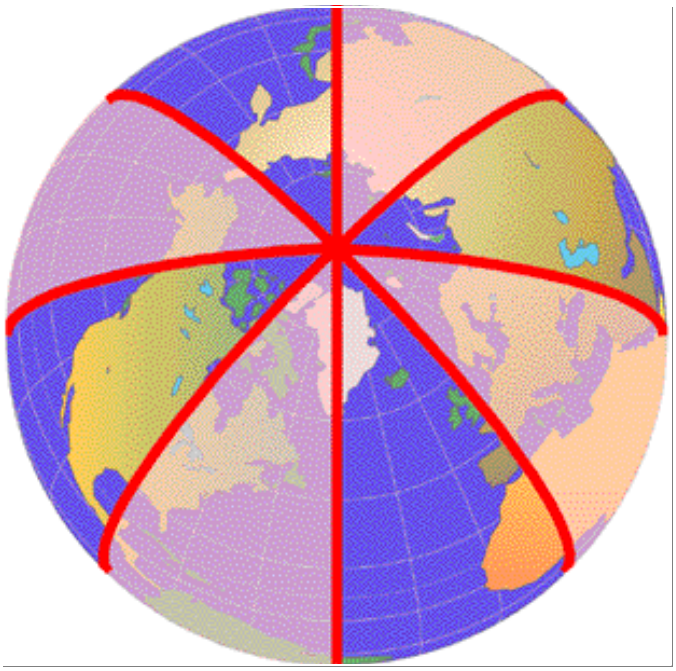


Figure 3-7 Longitude scrambling is used to reduce load imbalances due to computational differences stemming from day night cycles in a global NWP model. In this case, the model is run using 2 processes. One process has the brightly covered segments; the other has the darker colored segments. The effect is to give each process half the day-time points and half the night-time points.

The CREATE_DECOMP directive supports a feature called index scrambling that can reduce the effects of such load imbalances. Index scrambling is only allowed when there are no adjacent dependencies in the dimension to be scrambled because "EXCHANGE" communication would be very expensive if indices were scrambled. Several types of scrambling are supported. These include longitude scrambling to balance day/night load and latitude scrambling to balance winter/summer load. Both of these scrambling methods are useful in global NWP models.

To use index scrambling, a fourth parameter is added to the CREATE_DECOMP as shown in the code fragments below:

```
CSMS$CREATE_DECOMP(DECOMP_J, <jm>, <0>, <SCRAMBLE_LAT_STRATEGY>)

CSMS$CREATE_DECOMP(DECOMP_I, <im>, <0>, <SCRAMBLE_LON_STRATEGY>)
```

In the first case, parameter ***<SCRAMBLE_LAT_STRATEGY>*** indicates that the first decomposed dimension of ***DECOMP_J*** will be scrambled using a method appropriate for balancing load among latitudes in a global model. In the second case, parameter ***<SCRAMBLE_LON_STRATEGY>*** indicates that the first decomposed dimension of ***DECOMP_I*** will be scrambled using a method appropriate for balancing load among longitudes in a global model. (Note that neither decomposition has halo regions.) No other code changes are required

to use the scrambling feature. For this reason, it is convenient to add this feature as a performance optimization once debugging of the non-scrambled parallel code is complete.

3.5 More About DISTRIBUTE

The DISTRIBUTE directive will ignore scalar variables such as integer *avg* in following code fragment:

```
CSMS$DISTRIBUTE(DECOMP_I, <im>) BEGIN
      integer w(im), avg
CSMS$DISTRIBUTE END
```

The DISTRIBUTE directive will not change the declaration of *avg* because *avg* does not have a dimension of size *im* in its declaration. Also, *avg* will be treated as non-decomposed (duplicated on each process) by the other SMS directives. The behavior is the same as if the directive and declarations had been written like this:

```
CSMS$DISTRIBUTE(DECOMP_I, <im>) BEGIN
      integer w(im)
CSMS$DISTRIBUTE END
      integer avg
```

The DISTRIBUTE directive can decompose several types of arrays as shown the in the following code fragments:

```
CSMS$DISTRIBUTE(DECOMP_IJ, <im>, <jm>) BEGIN
      integer x(im,jm,km)
CSMS$DISTRIBUTE END
```

Here, the first dimension of array *x* is decomposed as described by the first decomposed dimension of ***DECOMP_IJ*** and the second dimension of array *x* is decomposed as described by the second decomposed dimension of ***DECOMP_IJ***. The third dimension of array *x* is not decomposed.

```
CSMS$DISTRIBUTE(DECOMP_IJ, <im>, <jm>) BEGIN
      real a(im,km,jm)
CSMS$DISTRIBUTE END
```

Here, the first dimension of array *a* is decomposed as described by the first decomposed dimension of ***DECOMP_IJ*** and the third dimension of array *a* is decomposed as described by the second decomposed dimension of ***DECOMP_IJ***. The second dimension of array *a* is not decomposed.

```
CSMS$DISTRIBUTE(DECOMP_IJ, <im>, <jm>) BEGIN
      real b(km,jm,im)
CSMS$DISTRIBUTE END
```

Here, the third dimension of array ***b*** is decomposed as described by the first decomposed dimension of ***DECOMP_IJ*** and the second dimension of array ***b*** is decomposed as described by the second decomposed dimension of ***DECOMP_IJ***. The first dimension of array ***b*** is not decomposed.

```
CSMS$DISTRIBUTE(DECOMP_IJ, <im>, <jm>) BEGIN
      real c(im,2,km)
CSMS$DISTRIBUTE END
```

Here, the first dimension of array ***c*** is decomposed as described by the first decomposed dimension of ***DECOMP_IJ***. The second and third dimensions of array ***c*** are not decomposed.

```
CSMS$DISTRIBUTE(DECOMP_IJ, <im>, <jm>) BEGIN
      real d(10,km)
CSMS$DISTRIBUTE END
```

Here, array ***d*** is not decomposed.

```
CSMS$DISTRIBUTE(DECOMP_IJ, <im>, <jm>) BEGIN
      real e(jm)
CSMS$DISTRIBUTE END
```

Here, the single dimension of array ***e*** is decomposed as described by the second decomposed dimension of ***DECOMP_IJ***.

All of the above declarations could equivalently be enclosed in one DISTRIBUTE directive pair as shown below:

```
CSMS$DISTRIBUTE(DECOMP_IJ, <im>, <jm>) BEGIN
      integer x(im,jm,km)
      real a(im,km,jm), b(km,jm,im), c(im,2,km), d(10,km), e(jm)
CSMS$DISTRIBUTE END
```

These simple examples obscure a few subtle features of parameters ***<im>*** and ***<jm>*** in the DISTRIBUTE directive. We have described these parameters as "array dimensions", but they are really somewhat more general. Consider the following code fragments:

```
CSMS$CREATE_DECOMP(DECOMP_IJ, <nx+2, ny+2>, <0,0>)
...
CSMS$DISTRIBUTE(DECOMP_IJ, <nx>, <ny>) BEGIN
      real u(nx+2,ny+2,nz)
CSMS$DISTRIBUTE END
```

These DISTRIBUTE directives will correctly translate declarations of array *u* in a manner analogous to the translation of array *x* in the previous example. However, notice that the second parameter is ***<nx>*** instead of ***<nx+2>*** as one might suspect. The string inside the angle brackets, *nx*, is really just used to identify array dimensions. This string is called a "dimension tag". The decoupling of "dimension tag" from the exact declared array dimensions provides some additional flexibility that minimizes the number of DISTRIBUTE directives that need to be used.

The dimension tags can be more complicated if necessary. For example, consider the following fragments from a program that uses dynamic memory:

```
[program main]

CSMS$CREATE_DECOMP(DECOMP_IJ, <nx+2, ny+2>, <0,0>)
      nxp2 = nx+2
      nyp2 = ny+2
...

[subroutine sub1]

CSMS$DISTRIBUTE(DECOMP_IJ, <nx,nxp2>, <ny,nyp2>) BEGIN
      real u(nx+2,ny+2,nz), a(nxp2,nyp2,nz)
CSMS$DISTRIBUTE END
```

Now the second parameter ***<nx,nxp2>*** has two tags, *nx* and *nxp2*. This indicates that array dimensions identified by either *nx* or *nxp2* will be decomposed as described by the first decomposed dimension of ***DECOMP_IJ***. Here, arrays *u* and *a* will be handled in exactly the same way during translation. The ability to specify more than one dimension tag for each decomposed dimension minimizes the number of DISTRIBUTE directives required in cases like this.

3.6 More About PARALLEL

There is no run-time performance penalty for using a PARALLEL directive because processes are not synchronized. Also, PARALLEL directives may enclose any valid Fortran executable statements. Therefore, a program that has only one decomposition will usually require no more than one BEGIN-END pair of PARALLEL directives for each program unit (subroutine, function, or main program).

The PARALLEL directive will translate serial loops correctly provided the upper and lower loop bounds are valid global indices. For example, the *i* and *j* loops below would all be correctly translated:

```
CSMS$PARALLEL(DECOMP_IJ,<i>,<j>) BEGIN
      do 100 k=1,km
      do 200 j=3,jm-2
      do 200 i=3,im-2
        z(i,j,k) = x(i,j,k) + y(i,j,k)
  200 continue

      do 210 j=1,2
      do 210 i=1,im
        z(i,j,k) = 0
  210 continue

      do 220 j=jm-1,jm
      do 220 i=1,im
        z(i,j,k) = 0
  220 continue

      do 230 j=1,jm
      do 230 i=1,2
        z(i,j,k) = 0
  230 continue

      do 240 j=1,jm
      do 240 i=im-1,im
        z(i,j,k) = 0
  240 continue

  100 continue
CSMS$PARALLEL END
```

In this code fragment, notice that translated loop 210 would only be executed on processes that contain global indices *j=1* or *j=2*. The PARALLEL directive ensures that other processes will skip loop 210. Similar translations will occur for the other loops.

It is useful to keep a few other caveats in mind when using the PARALLEL directive. Indices must be used consistently to avoid incorrect translation. Sometimes, indices are used for non-decomposed loops as well as for loops that span decomposed dimensions. This is the case in the following fragment:

```
CSMS$PARALLEL(DECOMP_IJ,<i>,<j>) BEGIN
      do 200 k=1,km
      do 200 j=1,jm
      do 200 i=1,im
        z(i,j,k) = x(i,j,k) + y(i,j,k)
  200 continue
      do 500 i=1,3
        call smooth(z)
```

```
  500 continue
CSMS$PARALLEL END
```

In this case, loop 500 is used to repeatedly call subroutine *smooth* which performs some computations on decomposed array *z*. This loop should NOT be translated because *i* is being used as an iteration count, not as an index into a decomposed dimension. This is easily fixed either by using a different loop index in loop 500, by moving the PARALLEL END directive to exclude loop 500, or by using the IGNORE directive as shown in Section 8.

Finally, it is almost always necessary to make sure that any loops containing decomposed arrays be enclosed inside PARALLEL directives. (A counter-example is described in the discussion of the TO_LOCAL directive in Section 4.) During translation, PPP will generate a warning message whenever it finds a loop that is not enclosed by PARALLEL directives if that loop contains a decomposed array. For example, suppose that we comment out the PARALLEL BEGIN (line 17) and PARALLEL END (line 29) directives in Example 3-1 (page 37).

```
C CSMS$PARALLEL(DECOMP_IJ,<i>,<j>) BEGIN
...
C CSMS$PARALLEL END
```

Assume the new program is stored in a file named *decomp_ex5.f*. The "Verbose" option of PPP, discussed in Section 11, can be used to cause warning messages to be displayed during translation:

```
>> ppp --Verbose=2 --Finclude=decomp_ex1.inc decomp_ex5.f
```

When the erroneous code is translated, the following warning message will be printed:

```
"./decomp_ex5_sms.f.tmp" 24 9 WARNING: This variable, decomposed by
CSMS$DISTRIBUTE, is being used outside of a parallel region.
```

If the program is built and run (ignoring the warning message), the following will appear on the screen:

```
>> smsRun 1 decomp_ex5_sms
 zsum =  300
>> smsRun 4 decomp_ex5_sms
 im =  15   jm =  10   km =  2
MPI: MPI_COMM_WORLD rank 1 has terminated without calling MPI_Finalize()
MPI: aborting job
< core dump >
```

What happened? With the PARALLEL directive removed, all loops remain un-translated and therefore span all global indices *i=1,15* and *j=1,10*. This was not a problem for the 1-process run because declarations remain full-sized. However, during the 4-process run, process-local array sizes are either *(8,5,2)* or *(7,5,2)* so the loops spanning *i=1,15* and *j=1,10* will go out of bounds. In the run shown above, the out of bounds writes cause a core dump.

However, behavior of any Fortran program that contains an out-of-bounds indexing bug can be very unpredictable and such bugs can be difficult to track down. It is best to use the "Verbose" option to PPP to generate warning messages and to check the code carefully any time this PPP warning message appears.

3.7 Arrays with Non-Unit Lower Bounds

Another issue to deal with regarding array declarations is the possibility that arrays may be declared with lower bounds other than one. For example, consider the following variant of Example 3-1:

```
[Include file:  decomp_ex6.inc]

      integer im, jm, km
      common /sizes_com/ im, jm, km
CSMS$DECLARE_DECOMP(DECOMP_IJ : <0,0>)

[Source file:  decomp_ex6.f]

      program decomp_ex6
      include 'decomp_ex6.inc'
      im = 15
      jm = 10
      km = 2
CSMS$CREATE_DECOMP(DECOMP_IJ, <im, jm>, <0,0>)
      call compute
      end

      subroutine compute
      include 'decomp_ex6.inc'
      integer i, j, k
CSMS$DISTRIBUTE(DECOMP_IJ, <im>, <jm>) BEGIN
      integer z(0:im-1,0:jm-1,0:km-1), zsum
CSMS$DISTRIBUTE END
CSMS$PARALLEL(DECOMP_IJ,<i>,<j>) BEGIN
      do 100 k=0,km-1
      do 100 j=0,jm-1
      do 100 i=0,im-1
        z(i,j,k) = 1
  100 continue
      zsum = 0

      do 200 k=0,km-1
      do 200 j=0,jm-1
      do 200 i=0,im-1
        zsum = zsum + z(i,j,k)
  200 continue

CSMS$PARALLEL END
CSMS$REDUCE(zsum, SUM)
      print *,'zsum = ',zsum
      return
```

```
      end
```

In this program array *z* is declared so the first index (lower bound) is zero in each dimension instead of the Fortran default of one. The bounds of loops 100 and 200 now start at zero. The only difference between the directives in this example and those in Example 3-1 is DECLARE_DECOMP. The new final parameter, ***<0,0>*** indicates that array declarations have a lower bound of zero in both decomposed dimensions. The colon ":" is used as a separator in this syntax so SMS won't confuse lower bounds with global arrays sized for a static memory case. For example, if we had accidentally used a comma "," instead of the colon, the directive would have looked like this:

```
C ERRONEOUS DIRECTIVE!
CSMS$DECLARE_DECOMP(DECOMP_IJ, <0,0>)
```

This would have been interpreted as a two-dimensional decomposition of statically allocated arrays with global sizes of zero in both decomposed dimensions! A correct way to mix static allocation and non-zero lower bounds is shown below:

```
CSMS$DECLARE_DECOMP(DECOMP_IJ, <im/2, jm/2> : <0,0>)
```

In this example, the second parameter represents local sizes (*<im/2, jm/2>*) and the third parameter is lower bound values (*<0,0>*) for the decomposition *DECOMP_IJ*.

4 Translating Array Indices

4.1 Translating Local Indices to Global Indices

When a loop has been translated using the PARALLEL directive, the value of the index is now process local as illustrated in Figure 2-2 and Figure 2-3. If the intent of the program is to access the global value, this index will need to be translated back to a global value. The TO_GLOBAL directive is used for this purpose as illustrated in Example 4-1.

```
[Include file:  tran_index.inc]

     1        integer im, jm
     2        common /sizes_com/ im, jm
     3  CSMS$DECLARE_DECOMP(DECOMP_IJ)

[Source file:  tran_index1.f]
      1       program tran_index1
      2       implicit none
      3       include 'tran1.inc'
      4       im = 5
      5       jm = 3
      6 CSMS$CREATE_DECOMP(DECOMP_IJ, <im, jm>, <0,0>)
      7       call compute
      8       end
      9
     10       subroutine compute
     11       implicit none
     12       include 'tran1.inc'
     13       integer i, j
     14 CSMS$DISTRIBUTE(DECOMP_IJ, <im>, <jm>) BEGIN
     15       integer x(im,jm)
     16 CSMS$DISTRIBUTE END
     17 CSMS$PARALLEL(DECOMP_IJ,<i>,<j>) BEGIN
     18       do 100 j=1,jm
     19       do 100 i=1,im
     20 CSMS$TO_GLOBAL(<1,i>, <2,j>) BEGIN
     21         x(i,j) = (100 * i) + j
     22 CSMS$TO_GLOBAL END
     23   100 continue
     24 CSMS$SERIAL BEGIN
     25       do j = 1, jm
     26         write(*,'(16i5)') (x(i,j),i=1,im)
     27       end do
     28 CSMS$SERIAL END
     29 CSMS$PARALLEL END
     30       return
     31       end
```

Example 4-1: An SMS parallel program that incorrectly initializes the array *x* inside subroutine *compute*.

This program initializes array *x* in loop 100 of subroutine *compute*. Each element of array *x* is then printed on the screen. When the serial code is run, the following is printed on the screen:

```
>> tran_index1
  101  201  301  401  501
  102  202  302  402  502
  103  203  303  403  503
```

Since *x(i,j) = (100 * i) + j*, each printed element appears as a three digit integer where the first digit is the *i* index, the second digit is "*0*", and the third digit is the *j* index. The same result is seen when the SMS version is run on one process. However, the results are incorrect when two processes are used:

```
>> smsRun 2 tran_index1_sms
  101  201  301  101  201
  102  202  302  102  202
  103  203  303  103  203
```

Why are the results incorrect? The PARALLEL directive has translated the *i* and *j* indices used to compute *x* in loop 100 using local indices. However, correct operation requires that *x* be initialized using global indices as in the original serial code. The solution is to use the TO_GLOBAL directive to translate the local indices to global indices. In this case, the body of loop 100 (line 18) would be replaced with the following code:

```
CSMS$TO_GLOBAL(<1,i>, <2,j>) BEGIN
        x(i,j) = (100 * i) + j
CSMS$TO_GLOBAL END
```

The first parameter in the TO_GLOBAL directive, ***<1,i>,*** indicates that array index *i* is an index in the first decomposed dimension. The second parameter, ***<2,j>,*** indicates that array index *j* is an index in the second decomposed dimension. All occurrences of indices *i* and *j* inside the TO_GLOBAL directives that are not array references will be converted to their global equivalents in the first and second decomposed dimensions, respectively.

Note that the TO_GLOBAL does not need an SMS decomposition name when it is enclosed by PARALLEL directives. In this case, TO_GLOBAL uses the decomposition specified by the enclosing PARALLEL directives. Directives TO_LOCAL and GLOBAL_INDEX, introduced later in this section, also have this feature. Running the new parallel code on various numbers of processes will now yield the same result as the serial run. Also note that since *p* is decomposed, the SERIAL directive is required to handle the print statement on line 26 as will be explained in Section 8.1.

The TO_GLOBAL directive is also commonly used in "*if*" statements such as the one shown below in loop 200:

```
 1        subroutine compute
 2        include 'tran_index.inc'
 3        integer i, j
 4   CSMS$DISTRIBUTE(DECOMP_IJ, <im>, <jm>) BEGIN
 5        integer x(im,jm)
 6   CSMS$DISTRIBUTE END
 7   CSMS$PARALLEL(DECOMP_IJ,<i>,<j>) BEGIN
 8        do 100 j=1,jm
 9        do 100 i=1,im
10   CSMS$TO_GLOBAL(<1,i>, <2,j>) BEGIN
11          x(i,j) = (100 * i) + j
12   CSMS$TO_GLOBAL END
13     100 continue
14        do 200 j=1,jm
15        do 200 i=1,im
16   CSMS$TO_GLOBAL(<1,i>) BEGIN
17          if (i.gt.3) then
18   CSMS$TO_GLOBAL END
19            x(i,j) = 0
20          endif
21     200 continue
22        call print_all(x)
23   CSMS$PARALLEL END
24        return
```

Example 4-2: A program that illustrates application of TO_GLOBAL to "if" statements.

Assume the new program is stored in a file named *tran_index2.f*. In the *if* statement on line 17, index *i* is compared with global index *3*. However, the enclosing PARALLEL directive will cause *i* to be translated to a local index. The TO_GLOBAL directive will cause *i* to be translated back to a global index for correct comparison with global index *3*. The output below shows that values of x are indeed set to zero for values of global index *i* greater than *3*:

```
>> smsRun 4 tran_index2_sms
  101  201  301    0    0
  102  202  302    0    0
  103  203  303    0    0
```

4.2 Translating Global Indices to Local Indices Inside Loops

Sometimes, indices that have been translated to global values need to be translated back to local values to be used as indices into decomposed arrays. The TO_LOCAL directive is used for this translation. Consider the following code fragment that uses computed indices to avoid out-of-bounds references:

```
      do 300 j=1,jm
      do 300 i=1,im
        im1 = max( 1,i-1)
        ip1 = min(im,i+1)
        x(i,j) = y(i,j) - y(im1,j) - y(ip1,j)
  300 continue
```

The max and min functions use index ***i*** in a comparison with global index values ***1*** and ***im***. Therefore, the TO_GLOBAL directive must be used (assume that the code fragment below is enclosed by a pair of PARALLEL directives):

```
      do 300 j=1,jm
      do 300 i=1,im
CSMS$TO_GLOBAL(<1,i>) BEGIN
        im1 = max( 1,i-1)
        ip1 = min(im,i+1)
CSMS$TO_GLOBAL END
        x(i,j) = y(i,j) - y(im1,j) - y(ip1,j)
  300 continue
```

The TO_GLOBAL directive will convert ***i-1*** and ***i+1*** to global values so ***ip1*** and ***im1*** will be computed as global indices. However, ***ip1*** and ***im1*** are then used as indices into decomposed array ***x***, so they must be converted back from global to local values to avoid out-of-bounds array references for multi-process runs. The TO_LOCAL directive is used to accomplish this as shown below:

```
      do 300 j=1,jm
      do 300 i=1,im
CSMS$TO_GLOBAL(<1,i>) BEGIN
CSMS$TO_LOCAL(<1,im1,ip1>) BEGIN
        im1 = max( 1,i-1)
        ip1 = min(im,i+1)
CSMS$TO_LOCAL END
CSMS$TO_GLOBAL END
        x(i,j) = y(i,j) - y(im1,j) - y(ip1,j)
  300 continue
```

Here, the TO_LOCAL and TO_GLOBAL directives are used in combination to accomplish both phases of index translation. The first parameter in the TO_LOCAL directive, ***<1,im1,ip1>,*** indicates that array indices ***im1*** and ***ip1*** are both used in loops that span the first decomposed dimension. In this example, occurrences of either index in code enclosed by the TO_LOCAL directives that are not array references will be converted to their local equivalents in the first decomposed dimension.

Sometimes, array indices are stored for later use. If conversion to local indices can be made before storage, then no index translation directives are required. This is the case in the following example:

```
[Include file:  tran_index3.inc]

 1       integer im
 2       common /sizes_com/ im
 3 CSMS$DECLARE_DECOMP(DECOMP_I)
 4
 5
[Source file:  tran_index3.f]
```

```
 1       program tran_index3
 2       include 'tran_index3.inc'
 3       im = 5
 4 CSMS$CREATE_DECOMP(DECOMP_I, <im>, <0>)
 5       call compute
 6
 7
 8       subroutine compute
 9       include 'tran_index3.inc'
10 CSMS$DISTRIBUTE(DECOMP_I, <im>) BEGIN
11       integer x(im), i, pack_num, ip
12       integer xpack(im), i_pack(im)
13 CSMS$DISTRIBUTE END
14 CSMS$PARALLEL(DECOMP_I,<i>) BEGIN
15       do 100 i=1,im
16 CSMS$TO_GLOBAL(<1,i>) BEGIN
17         x(i) = 100 * i
18 CSMS$TO_GLOBAL END
19   100 continue
20       pack_num = 0
21       do 400 i=1,im
22         if (x(i).gt.300) then
23           pack_num = pack_num + 1
24           xpack(pack_num) = x(i)
25           i_pack(pack_num) = i
26         endif
27   400 continue
28 CSMS$PARALLEL END
29       call pack_compute(xpack,pack_num)
30       do 500 ip=1,pack_num
31         x(i_pack(ip)) = xpack(ip)
32   500 continue
33       print *,'ARRAY x:'
34 CSMS$SERIAL BEGIN
35       write(*, '(5I5)') (x(i), i=1,im)
36 CSMS$SERIAL END
37       return
38       end
39
40       subroutine pack_compute(xp,pnum)
41       integer pnum, xp(pnum), p
42       do 600 p=1,pnum
43         xp(p) = 0
44   600 continue
45       return
46       end
```

Example 4-3: This program illustrates indirect indexing. No directives are required in subroutine *pack_compute*.

Here, subroutine *compute* initializes decomposed array *x* and then "packs" selected values of *x* into array *xpack* for further processing by subroutine *pack_compute*. Indices of selected values are stored in array *i_pack*. After the selected values of *x* are modified by

pack_compute, they are "unpacked" back into array *x*. Loop 400 does the selection and packing and loop 500 does the unpacking. The SERIAL directive will be explained in Section 8.1. When the serial code is run, the following output is printed on the screen:

```
>> tran_index3
 ARRAY x:
  100  200  300    0    0
```

In this example, the computations inside subroutine pack_compute are very simple: each packed data point is just set to zero. Running the parallel code on different numbers of processes yields the same results.

Subroutine *pack_compute* has no computational dependencies (it is "embarrassingly parallel"). As a result, no SMS directives are required. This type of packing and unpacking is common in NWP models, especially in physics subroutines. In fact, subroutines like *pack_compute* may call many other subroutines in the same fashion, with none of them requiring any SMS directives. It is not uncommon for large portions of a NWP model to require no SMS directives.

Note that loop 500 need not be enclosed inside the PARALLEL directives because loop index *ip* is purely local. If this code is translated using the --Verbose=2 option to PPP, the expected warning message appears because array *x* is being used in a loop that is not inside a parallel region:

```
"./tran_index3_sms.f.tmp" 32 25 WARNING: This variable, decomposed by
CSMS$DISTRIBUTE, is being used outside of a parallel region.
```

The warning message can be safely ignored in this case.

4.3 Using TO_LOCAL to Generate Processor Local Sizes and Loop Bounds

In many NWP models, large sections of code contain no dependencies that require communications (typically model physics routines). If the array bounds and loop limits are passed into these routines, SMS provides a means to parallelize them without inserting directives into the code. Example 4-4 shows such a case.

```
 1         program AVOID_DIRECTIVES
 2         implicit none
 3         include 'tran_index.inc'
 4         im = 8
 5         jm = 6
 6   CSMS$CREATE_DECOMP(DECOMP_IJ, <im, jm>, <2,2>)
 7         call compute
 8         end
 9
10         subroutine compute
11         implicit none
```

```
12        include 'tran_index.inc'
13        integer i, j
14        integer istart, iend, jstart, jend
15    CSMS$DISTRIBUTE(DECOMP_IJ, <im>, <jm>) BEGIN
16        integer x(im,jm), y(im,jm)
17    CSMS$DISTRIBUTE END
18
19    CSMS$PARALLEL(DECOMP_IJ,<i>,<j>) BEGIN
20
21        do 100 j=1,jm
22        do 100 i=1,im
23    CSMS$TO_GLOBAL(<1,i>, <2,j>) BEGIN
24          x(i,j) = (100 * i) + j
25    CSMS$TO_GLOBAL END
26      100 continue
27
28        y = 0.0
29
30        istart = 1
31        iend   = im - 1
32        jstart = 2
33        jend   = jm
34
35    csms$to_local(<1, im     : size  >, <2, jm   : size  >,
36    csms$>        <1, istart : lbound>, <1, iend : ubound>,
37    csms$>        <2, jstart : lbound>, <2, jend : ubound>) begin
38        call physics(x, im, jm, istart, iend, jstart, jend, y)
39    csms$to_local end
40    CSMS$SERIAL BEGIN
41        do j = 1, jm
42          write(*,'(16i5)') (y(i,j),i=1,im)
43        end do
44    CSMS$SERIAL END
45    CSMS$PARALLEL END
46        return
47        end
48
49
50
51        subroutine physics(arr_in, dim1_size, dim2_size,
52       &                   dim1_start, dim1_end,
53       &                   dim2_start, dim2_end,
54       &                   arr_out)
55        integer dim1_size, dim2_size
56        integer arr_in(dim1_size, dim2_size)
57        integer dim1_start, dim1_end
58        integer dim2_start, dim2_end
59        integer arr_out(dim1_size, dim2_size)
60
61        integer i, j
62        do j = dim2_start, dim2_end
63          do i = dim1_start, dim1_end
64            arr_out(i,j) = 2.0*arr_in(i,j)
65          end do
```

```
66          end do
67          return
68          end

48
```

Example 4-4 Sample code that shows how TO_LOCAL can be used to pass local array bounds and start/end loop limits to subroutines so that no directives be added to the called routines.

Program *AVOID_DIRECTIVES* calls subroutine *physics* (line 38), passing the arrays *x* and *y*, the sizes for each dimension (*im* and *jm*) and the starting and ending loop limits (*istart*, *iend, jstart, jend*) over which the loops in physics will span. The TO_LOCAL directive at lines 35-37 converts the dimensions and loop limits to their process local values. The syntax

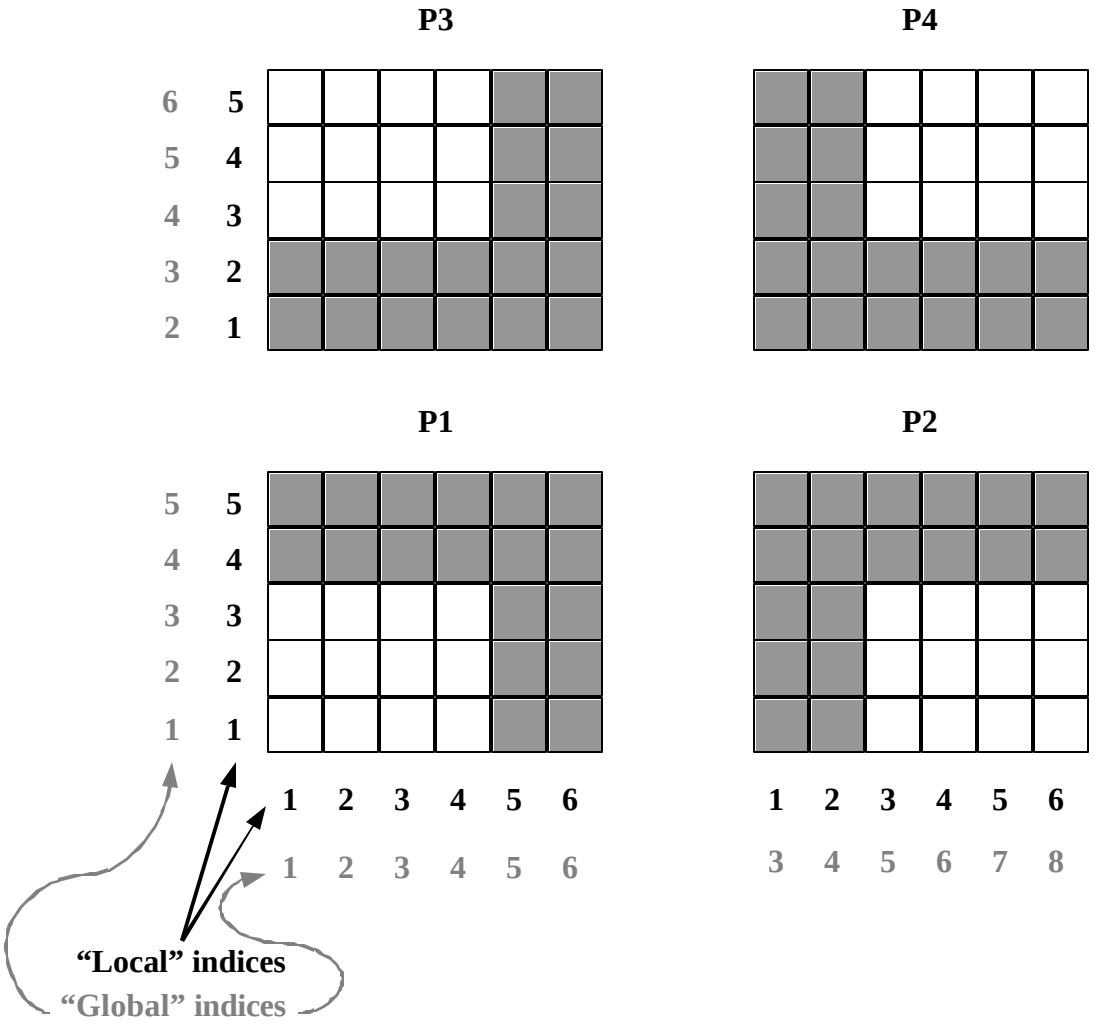
<1, im : size>

causes PPP to replace references to *im* with the process local size for the first decomposed dimension, (where the size includes the number of halo points). For a static memory model, the size would be the local size declared in the DECLARE_DECOMP directive. The syntax

<1, istart : lbound>

causes PPP to replaced instances of *istart* with the local index of the first interior point for the first decomposed dimension for the given process. Figure 4-1 shows all the sizes and bounds for this case, assuming the program is run on 4 processes.

The result is that, inside subroutine physics, the *dim1_size, dim2_size, dim1_start, dim1_end, dim2_start, and dim2_end* all have the correct process local values. Consequently, subroutine *physics* produces the right answer for any process decomposition, even though it contains no SMS directives.



Processor	Decomposed dimension	Size	Lbound	Ubound
P1	1	6	1	4
P1	2	5	1	3
P2	1	6	3	6
P2	2	5	1	3
P3	1	6	1	4
P3	2	5	3	5
P4	1	6	3	6
P4	2	5	3	5

Figure 4-1 Process layout, local sizes, lower bounds and upper bounds for a 4 process run of Example 4-4.

4.4 Global-to-Local Index Translation with Restricted Execution

The form of the TO_LOCAL directive described above should always be used in combination with a TO_GLOBAL directive. Otherwise, there will be no assurance that the global index being translated actually belongs on a process. For example, consider the following code fragment that is enclosed in a PARALLEL directive but is not inside a loop:

```
      id = 5
      jd = 4
      x(id,jd) = 10
```

The following use of TO_LOCAL would be incorrect:

```
CSMS$TO_LOCAL(<1,id>,<2,jd>) BEGIN
      id = 5
      jd = 4
CSMS$TO_LOCAL END
      x(id,jd) = 10
```

The translation of ***id*** and ***jd*** from global values to process-local values will work fine on the process that "owns" global point ***(5,4)***. However, the translation will be erroneous on processes that do not own global point ***(5,4)*** because there is no valid local equivalent of these global coordinates on those processes. In order to restrict the execution of these statements to the process that owns the data, the GLOBAL_INDEX directive must be used as shown below:

```
      id = 5
      jd = 4
CSMS$GLOBAL_INDEX(1,2) BEGIN
      x(id,jd) = 10
CSMS$GLOBAL_INDEX END
```

The GLOBAL_INDEX directives perform the correct index translations AND ensure that the enclosed code is only executed on the process that owns global point ***(5,4)***. In this case, the first parameter in the directive, ***1***, indicates that all array indices corresponding to the first decomposed dimension will be translated to their local equivalents. The second parameter, ***2***, indicates that all array indices corresponding to the second decomposed dimension will be translated to their local equivalents. In addition, execution of the enclosed assignment statement will only be permitted on the process that contains global point ***(id,jd)***.

Consider the following example that initializes the boundaries of an array that is decomposed in two dimensions:

```
 1       subroutine compute
 2       include 'tran_index5.inc'
 3 CSMS$DISTRIBUTE(DECOMP_IJ, <im>, <jm>) BEGIN
 4       integer x(im,jm)
 5 CSMS$DISTRIBUTE END
```

```
 6       integer i, j
 7 CSMS$PARALLEL(DECOMP_IJ,<i>,<j>) BEGIN
 8       do 100 j=1,jm
 9       do 100 i=1,im
10 CSMS$TO_GLOBAL(<1,i>, <2,j>) BEGIN
11         x(i,j) = (100 * i) + j
12 CSMS$TO_GLOBAL END
13   100 continue
14       do 110 j=2,jm-1
15 CSMS$GLOBAL_INDEX(1) BEGIN
16         x( 1,j) = 0
17         x(im,j) = 0
18 CSMS$GLOBAL_INDEX END
19   110 continue
20       do 120 i=2,im-1
21 CSMS$GLOBAL_INDEX(2) BEGIN
22         x(i, 1) = 0
23         x(i,jm) = 0
24 CSMS$GLOBAL_INDEX END
25   120 continue
26 CSMS$GLOBAL_INDEX(1,2) BEGIN
27       x( 1, 1) = 0
28       x(im, 1) = 0
29       x( 1,jm) = 0
30       x(im,jm) = 0
31 CSMS$GLOBAL_INDEX END
32       print *,'ARRAY x:'
33       call print_all(x)
34 CSMS$PARALLEL END
35       return
36       end
```

Example 4-5: An SMS program that illustrates the use of the GLOBAL_INDEX directive to initialize boundaries.

This program initializes array *x* as in previous examples. It then proceeds to set the boundary values of *x* to zero in lines 14 through 30. Assume the new program is stored in a file named *tran_index5.f*. When the serial code is run, the following is printed on the screen:

```
>> tran_index5
 ARRAY x:
    0    0    0    0    0
    0  202  302  402    0
    0    0    0    0    0
```

Three pairs of GLOBAL_INDEX directives handle the necessary translations. The first pair deals with global indices *1* and *im* in loop 110 while the second pair deals with global indices *1* and *jm* in loop 120. The third pair handles global indices in the four assignment statements on lines 27 through 30. In each case, indices are translated and execution of each enclosed statement is permitted only on appropriate processes. When this program is run on multiple processes, the expected results are printed on the screen.

5 Handling Adjacent Dependencies

5.1 Further Details on EXCHANGE

In Section 2.5, we saw how the EXCHANGE directive was used to implement communications needed to resolve adjacent dependencies for a dynamic memory, one dimensional decomposition case where the halo regions required were of width 1. In this sub-section, we expand on that discussion by examining the treatment of two-dimensional decompositions, larger stencils, and by discussing other miscellaneous details about EXCHANGE.

5.1.1 Using EXCHANGE in the Case of Two-Dimensional Decompositions

We begin by modifying the Laplace Example 2-5 introduced in Section 2.5 so that a two dimensional decomposition is used. Two dimensional data decompositions allow parallel programs to scale to a large number of processes.

```
 1        program basic_ex_2d_decomp
 2        include 'basic.inc'
 3        im = 10
 4        jm = 10
 5  CSMS$CREATE_DECOMP(DECOMP_I, <im,jm>, <1,1>)
 6        call laplace
 7        end
 8
 9        subroutine laplace
10        include 'basic.inc'
11        integer i, j, iter
12        real max_error
13        real tolerance
14        parameter (tolerance = 0.001)
15  CSMS$DISTRIBUTE(DECOMP_I, <im>, <jm>) BEGIN
16        real f(im,jm), df(im,jm)
17  CSMS$DISTRIBUTE END
18  CSMS$PARALLEL(DECOMP_I,<i>, <j>) BEGIN
19        do 100 j=1,jm
20        do 100 i=1,im
21          f(i,j) = 0.0
22    100 continue
23        do 110 j=1,jm
24  CSMS$GLOBAL_INDEX(1) BEGIN
25          f( 1,j) = 2.0
26          f(im,j) = 2.0
27  CSMS$GLOBAL_INDEX END
28    110 continue
29        do 120 i=1,im
30  CSMS$GLOBAL_INDEX(2) BEGIN
```

```
31          f(i, 1) = 2.0
32          f(i,jm) = 2.0
33    CSMS$GLOBAL_INDEX END
34      120 continue
35          iter = 0
36          max_error = 2.0 * tolerance
37    C main iteration loop...
38          do while ((max_error .gt. tolerance) .and. (iter .lt. 1000))
39            iter = iter + 1
40            max_error = 0.0
41    CSMS$EXCHANGE(f)
42            do 200 j=2,jm-1
43            do 200 i=2,im-1
44              df(i,j) = 0.25*(f(i-1,j) + f(i+1,j) + f(i,j-1) + f(i,j+1))
45         &                  - f(i,j)
46      200 continue
47            do 300 j=2,jm-1
48            do 300 i=2,im-1
49              if (max_error .lt. abs(df(i,j))) then
50                max_error = abs(df(i,j))
51              endif
52      300 continue
53    CSMS$REDUCE(max_error, MAX)
54            do 400 j=2,jm-1
55            do 400 i=2,im-1
56              f(i,j) = f(i,j) + df(i,j)
57      400 continue
58          enddo
59
60    CSMS$PARALLEL END
61          print *, 'Solution required ',iter,' iterations'
62          print *, 'Final error = ', max_error
63
64          return
65          end
```

Example 5-1 Two-dimensional decomposition version of Example 2-5.

The CREATE_DECOMP directive now lists two decomposed dimension (with global sizes *im* and *jm*). The halo width for each dimension is 1 in this case. As discussed in Section 3.2, the DISTRIBUTE, PARALLEL, and GLOBAL_INDEX directives are modified to handle the 2-D decompositions. Although the communication patterns required to support 2-dimensional decompositions are more complex than the 1-dimensional case, SMS hides all of these details. Thus, the EXCHANGE directive is unchanged. Figure 5-1 shows some sample stencils overlaid on a 3x3 processor decomposition of the problem. The halo regions are the shaded areas. The white boxes are referred to as the "interior" of each process's sub-domain. The stencil centered at global coordinate *(3,2)* only requires P1 communicate with P2. However, the stencil centered at global coordinate *(4,4)* requires P5 communicate with both P2 and P4. Figure 5-2 and Figure 5-3 show the full communications pattern for a 2-D exchange. Notice that the corner halo points of the center process are filled with data from the corresponding corner processes in the drawing.

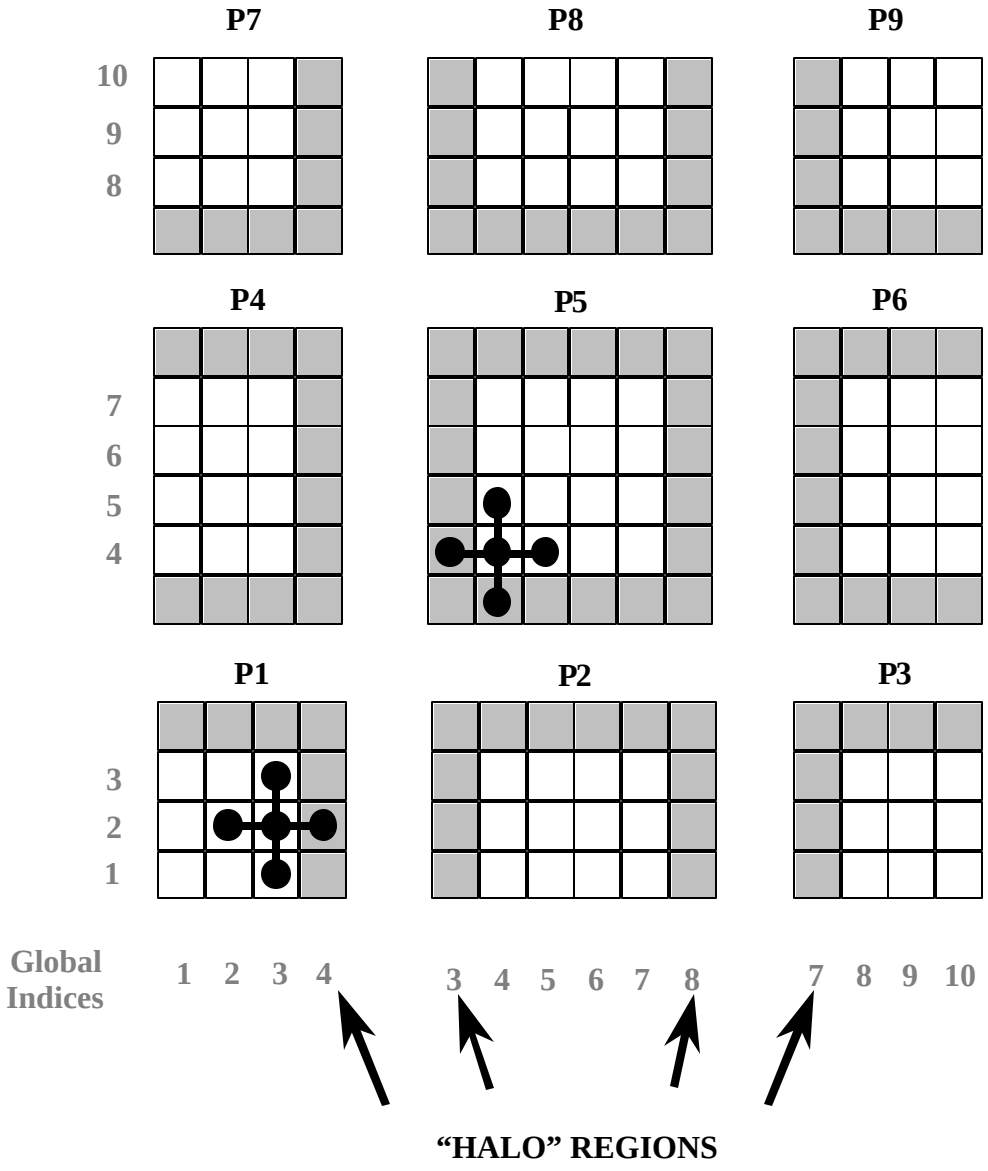


Figure 5-1 Sample stencils overlaid on a 3x3 process decomposition for the Laplace problem. The halo regions are the shaded areas. The white boxes are referred to as the "interior" of each process's sub-domain.

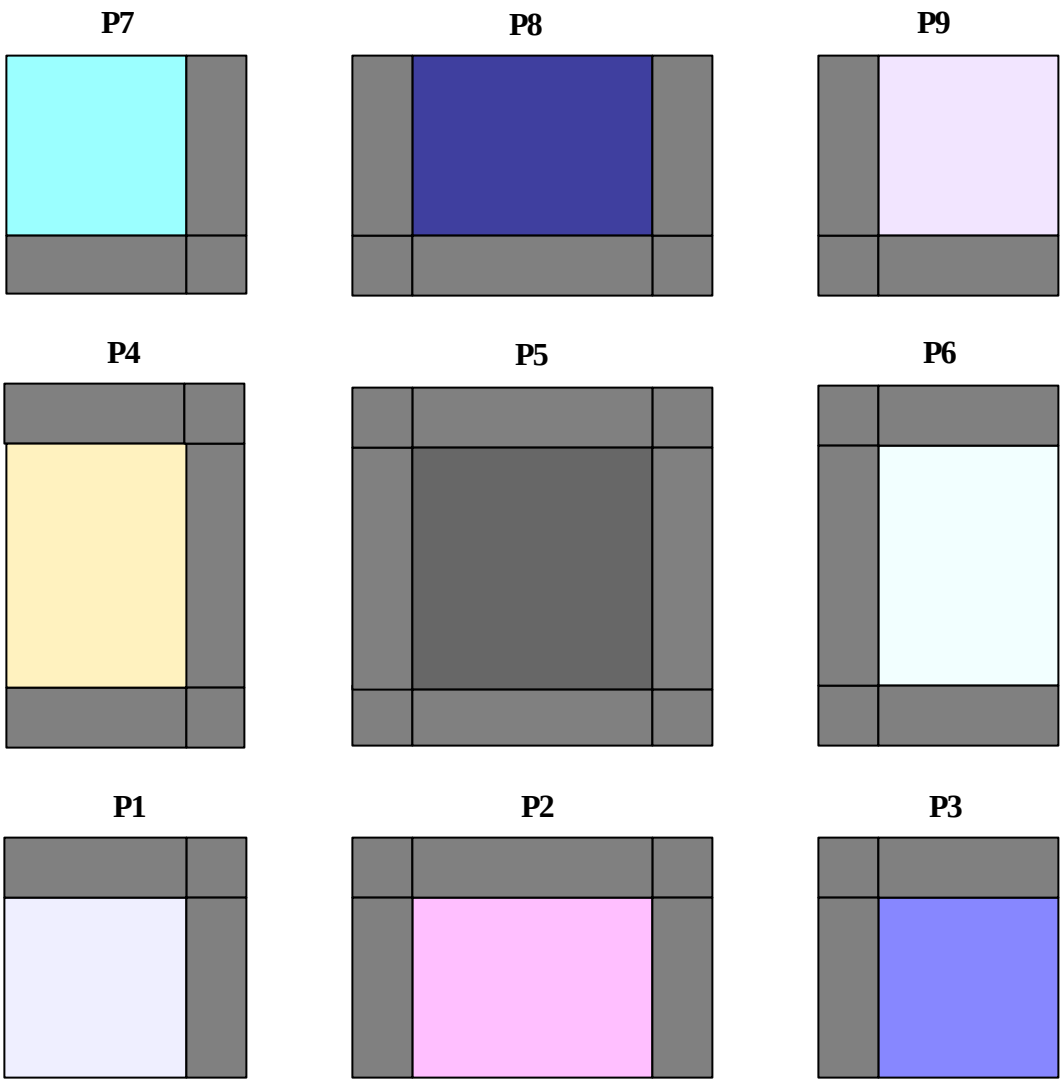


Figure 5-2 Schematic of how data are distributed among 9 processes just prior to an exchange operation. The big boxes contain the data. The boxes on the edges are the halo regions.

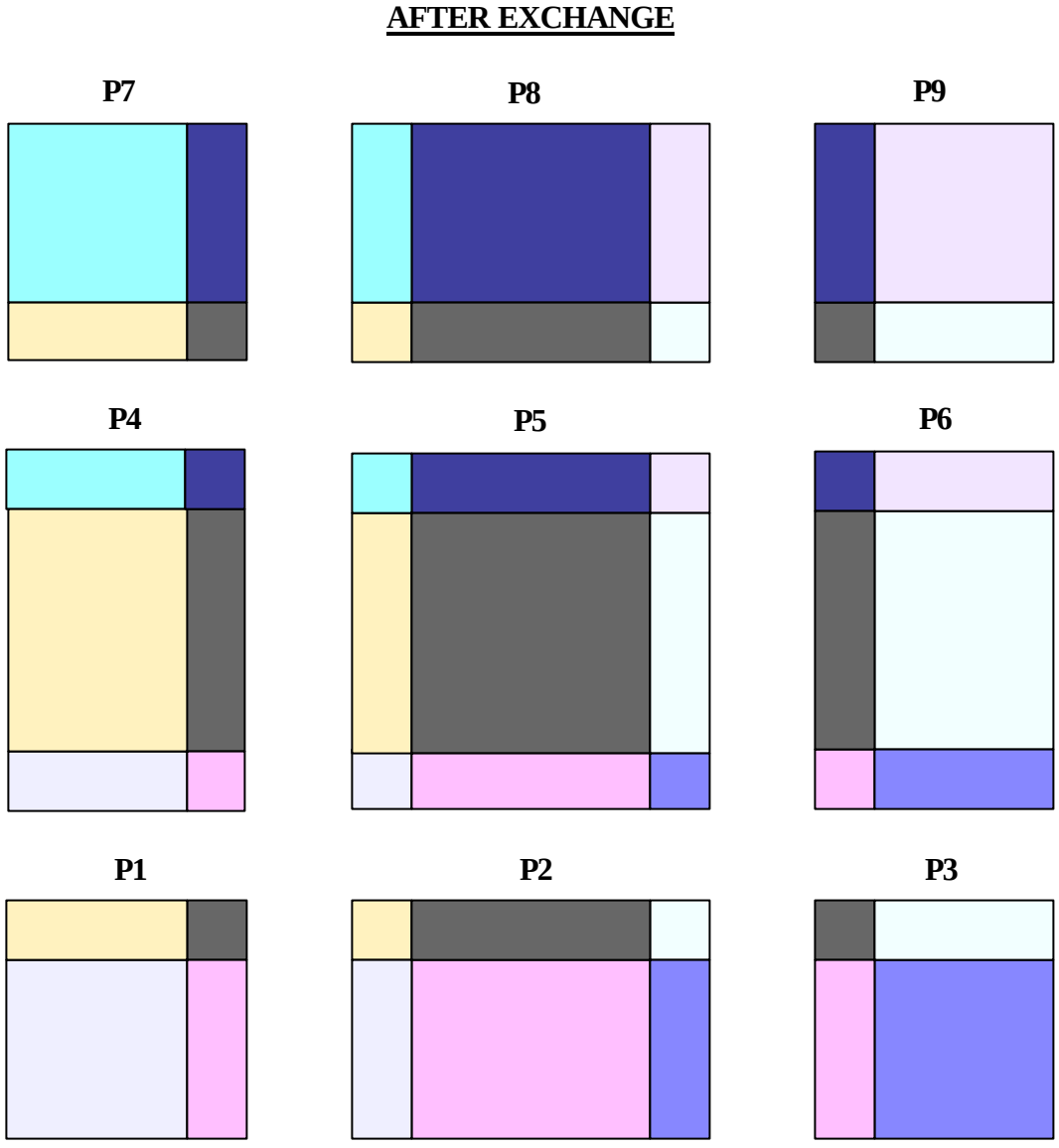


Figure 5-3 Illustration of the data distribution just after a 2 dimensional exchange. The halo regions in Figure 5-2 have been filled with the data from the corresponding neighboring processes.

The obvious cases when 2-D decompositions are required occur for problems having fewer points in a decomposed dimension than there are processors available. For instance, Example 2-5 (page 32) could run on at most 10 processes because the size of the decomposed dimension is 10. Another, more subtle, issue is that adjacent communication only scales when 2-D process layouts are used. Figure 5-4, Figure 5-5, Figure 5-6, and Figure 5-7 show why this is the case for exchanges made on a size 16x16 array.

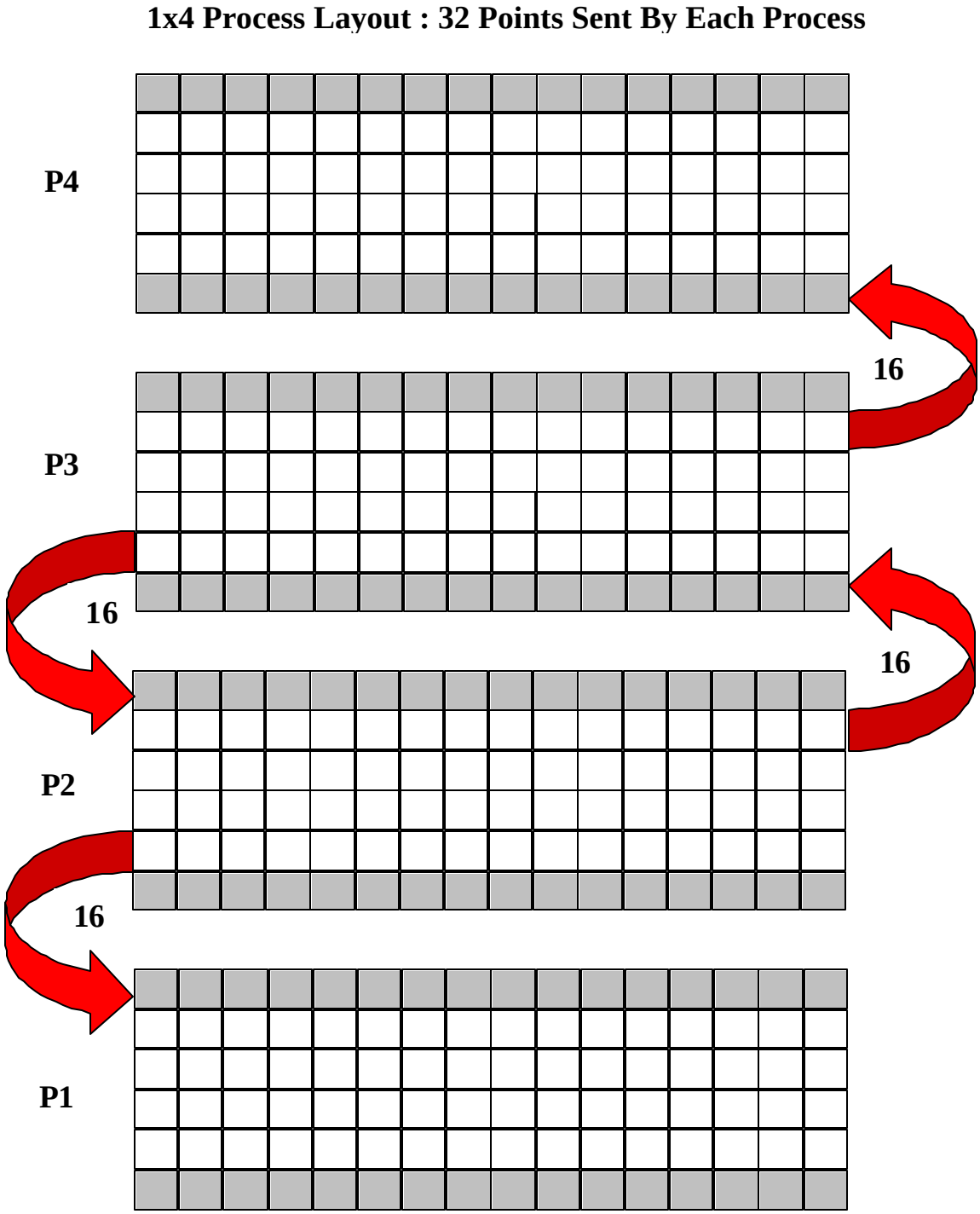


Figure 5-4 Schematic of the number of data points sent by each process during an exchange for a 1x4 process layout. In this case, each process sends 16 data points in each of 2 directions for a total of 32. In this figure and the three that following, edge processes include halo regions on both sides for illustration purposes even though SMS does not currently support periodic boundary conditions.

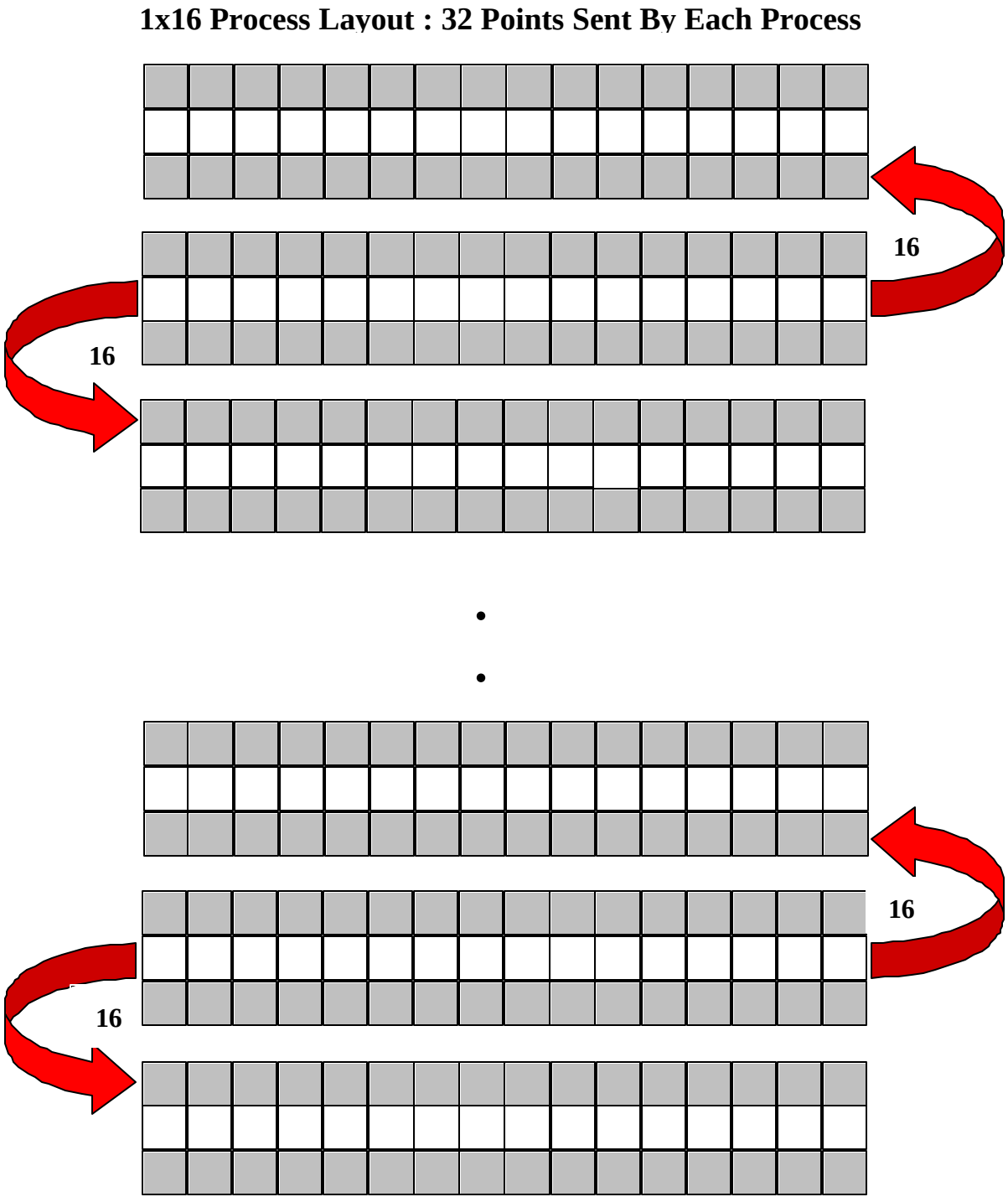


Figure 5-5 Schematic of the number of data points sent by each process during an exchange for a 1x16 process layout. In this case, each process sends 16 data points in each of 2 directions for a total of 32.

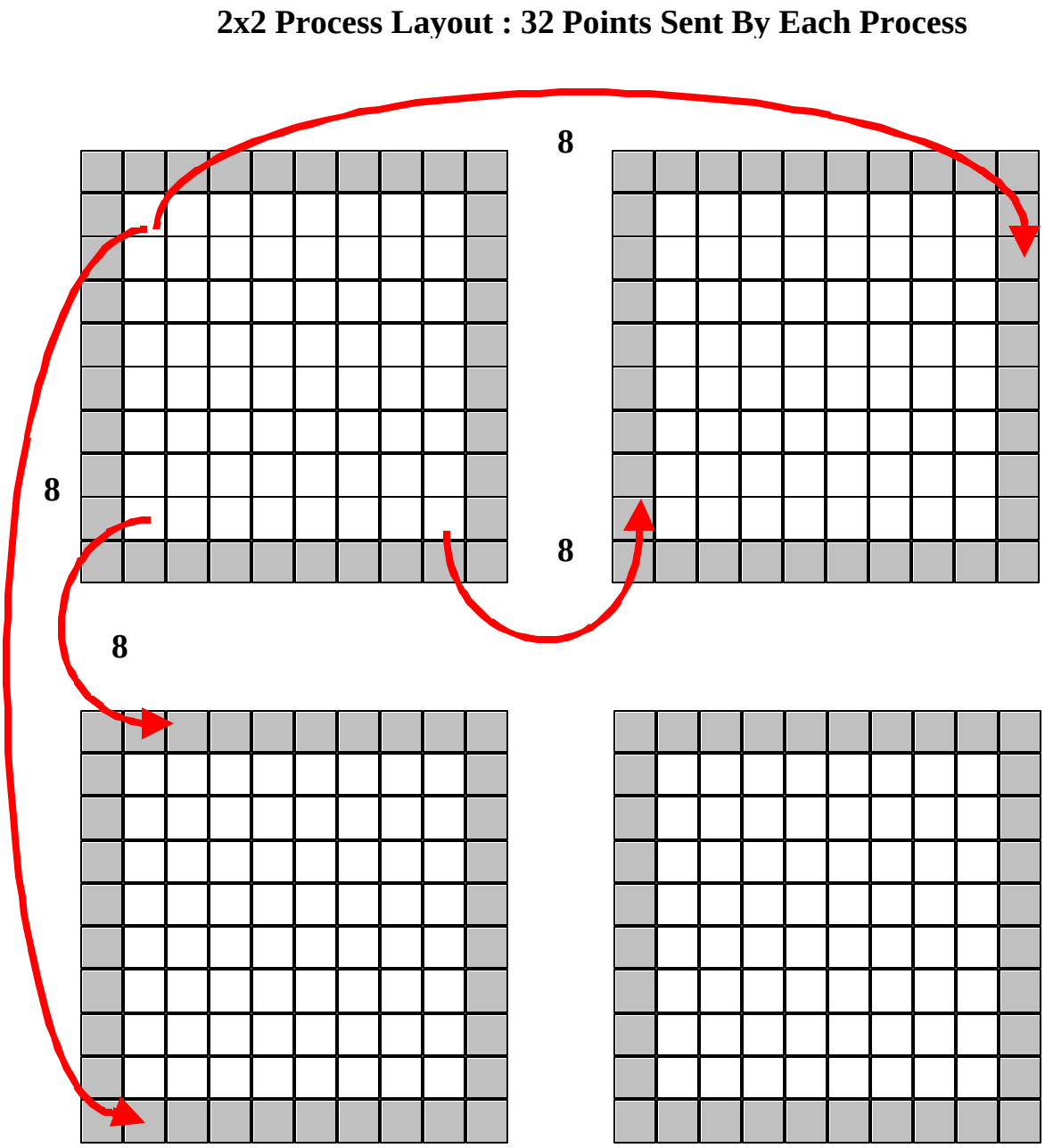


Figure 5-6 Schematic of the number of data points sent by each process during an exchange for a 2x2 process layout. In this case, each process sends 8 data points in each of 4 directions for a total of 32.

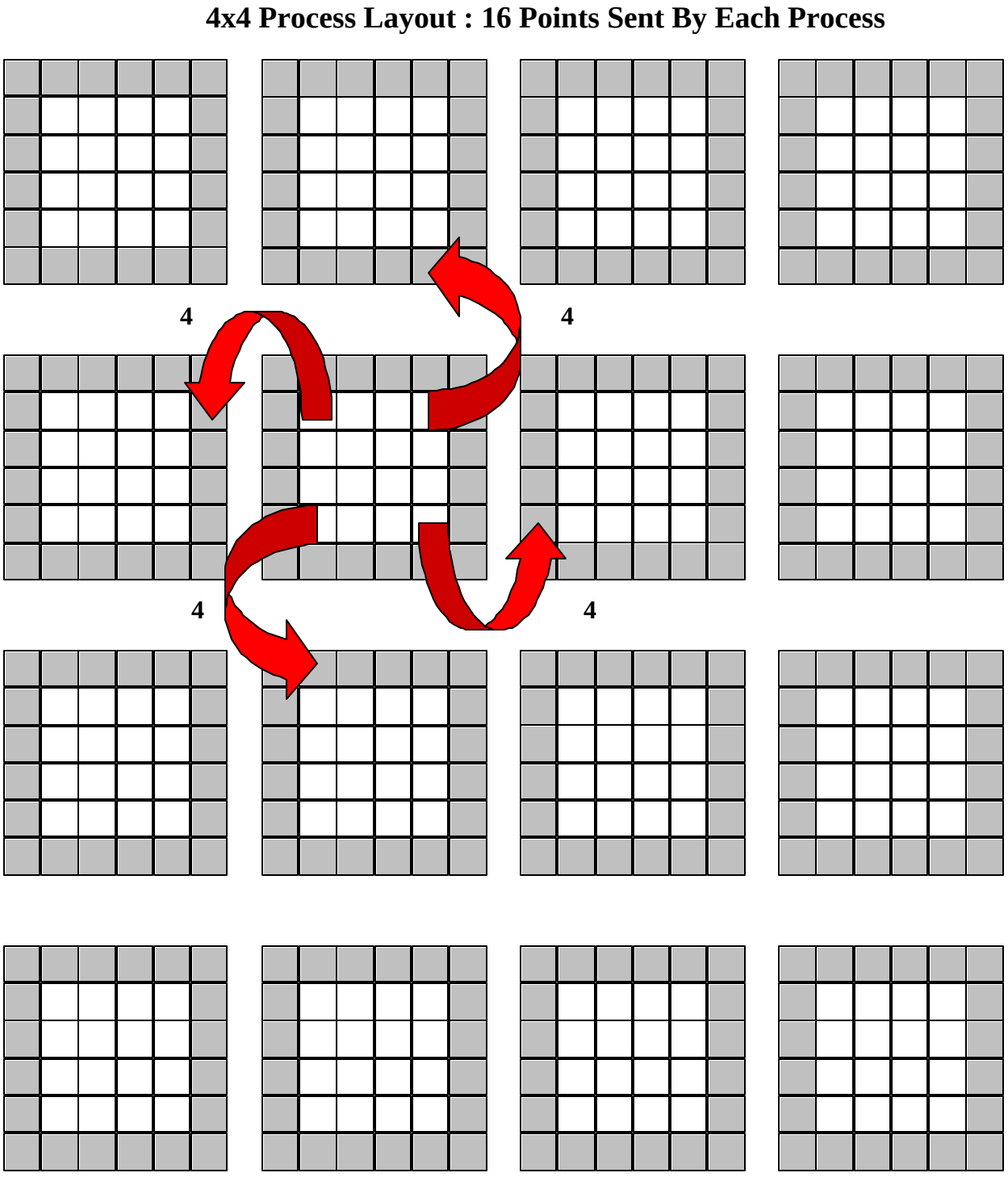


Figure 5-7 Schematic of the number of data points sent by each process during an exchange for a 4x4 process layout. In this case, each process sends 4 data points in each of 4 directions for a total of 16.

If only one dimension is decomposed, the number of data points exchanged between neighbors remains constant when the number of processes increases from 4 to 16 (and beyond) (Figure 5-4 and Figure 5-5). However, if a 2-D decomposition is implemented then when the process layout is changed from 2x2 (4 total) to 4x4 (16 total), the number of data points exchanged is halved (Figure 5-6 and Figure 5-7). The general rule is that if square process layouts are used, the number of data points communicated scales as $1/\sqrt{Np}$, where Np is the number of processes. As seen in Section 3.3.1, SMS tries to make the process layouts as close to square as possible.

5.1.2 Larger Stencils

As illustrated in Figure 2-14, the widths of the stencil for the calculation of df in the laplace program is one point in each direction. Since this is the only computation in Laplace requiring "exchange", it is clear that the halo widths specified in CREATE_DECOMP must be 1 in the i and j dimensions. However, suppose we modify Example 2-4 as shown in Example 5-2 below.

```
 1        program basic_ex_halo2
 2        include 'basic.inc'
 3        im = 10
 4        jm = 10
 5  CSMS$CREATE_DECOMP(DECOMP_I, <im>, <2>)
 6        call laplace
 7        end
 8
 9        subroutine laplace
10        include 'basic.inc'
11        integer i, j, iter
12        real max_error
13        real tolerance
14        parameter (tolerance = 0.001)
15  CSMS$DISTRIBUTE(DECOMP_I, <im>) BEGIN
16        real f(im,jm), df(im,jm)
17  CSMS$DISTRIBUTE END
18  CSMS$PARALLEL(DECOMP_I,<i>) BEGIN
19        do 100 j=1,jm
20        do 100 i=1,im
21          f (i,j) = 0.0
22          df(i,j) = 0.0
23    100 continue
24        do 110 j=1,jm
25  CSMS$GLOBAL_INDEX(1) BEGIN
26          f( 1,j) = 2.0
27          f(im,j) = 2.0
28  CSMS$GLOBAL_INDEX END
29    110 continue
30        do 120 i=1,im
31          f(i, 1) = 2.0
32          f(i,jm) = 2.0
33    120 continue
34        iter = 0
35        max_error = 2.0 * tolerance
```

```
36  C main iteration loop...
37        do while ((max_error .gt. tolerance) .and. (iter .lt. 1000))
38          iter = iter + 1
39          max_error = 0.0
40  CSMS$EXCHANGE(f)
41          do 200 j=2,jm-1
42          do 200 i=2,im-1
43            df(i,j) = 0.25*(f(i-1,j) + f(i+1,j) + f(i,j-1) + f(i,j+1))
44       &                - f(i,j)
45    200 continue
46          do 300 j=2,jm-1
47          do 300 i=2,im-1
48            if (max_error .lt. abs(df(i,j))) then
49              max_error = abs(df(i,j))
50            endif
51    300 continue
52  CSMS$REDUCE(max_error, MAX)
53          do 400 j=2,jm-1
54          do 400 i=2,im-1
55            f(i,j) = f(i,j) + df(i,j)
56    400 continue
57        enddo
58
59  CSMS$EXCHANGE(df)
60        do j = 1, jm
61          do i = 3, im-2
62            f(i,j) = f(i,j) + 2.0*df(i,j) - df(i-2,j) - df(i+2,j)
63          end do
64        end do
65
66  CSMS$PARALLEL END
67        print *, 'Solution required ',iter,' iterations'
68        print *, 'Final error = ', max_error
69
70        end
```

Example 5-2 Modified version of Example 2-5 with additional code that has a stencil width of 2 in the *i* direction.

For the calculations starting at line 60, the width of the stencil is 2 in the *i* direction and 1 in the *j* direction as shown in Figure 5-8.

```
df2(i,j) = 2.0*df(i,j) - df(i-2,j) - df(i+2,j)
```

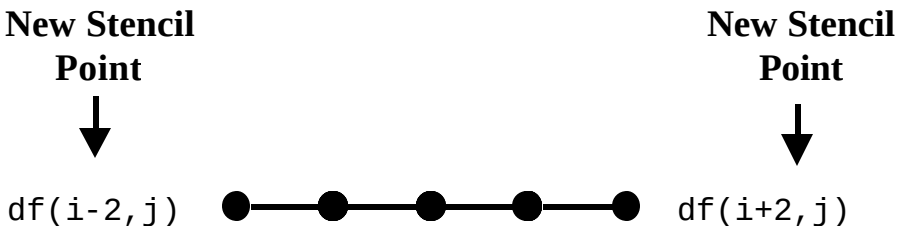


Figure 5-8 Modified stencil for additional calculations in Example 5-2. This time the stencil width is 2 in the *i* direction.

The exchanges of the size 2 halo regions are aggregated to reduce latency as shown in Figure 5-9.

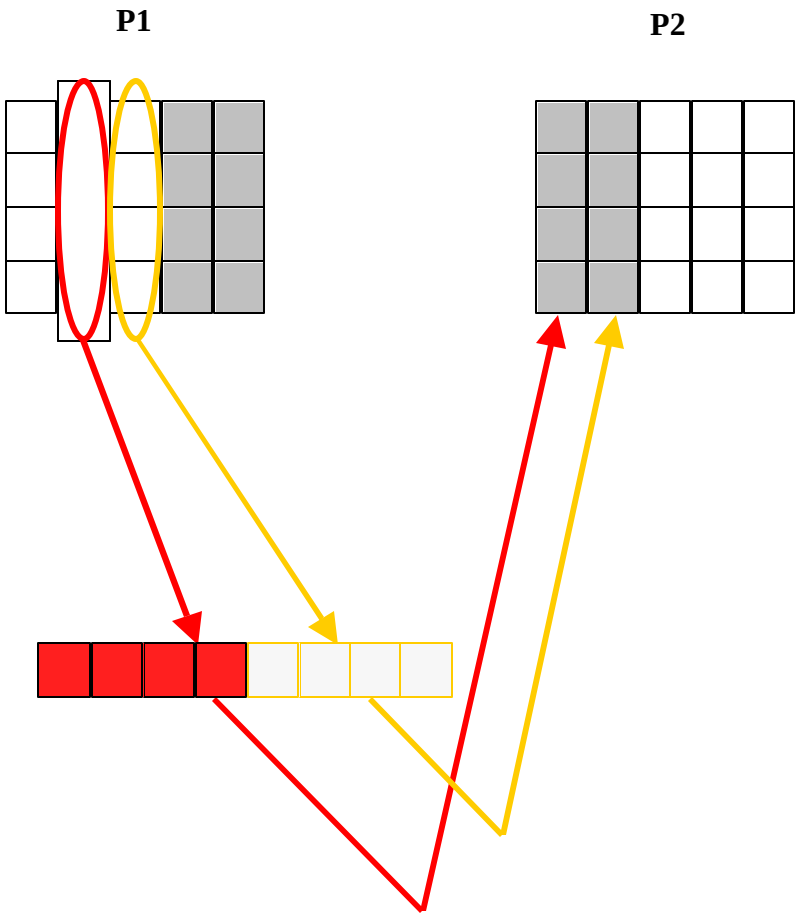


Figure 5-9 A illustration showing how data points from two-point thick halo regions are combined into a single message that is sent to the neighboring process in order to reduce latency.

This program now has 2 calculations involving the same dimension of the same decomposition with different stencil widths. SMS handles this by requiring the programmer to make the halo width of the decomposition equal to the larger of the two widths. It is up to the programmer to determine the width of the largest stencil required by each dimension of every decomposition. The CREATE_DECOMP directive (line 5), of Example 5-2 (page 78) shows the correct halo width specification (*<2>*).

Choosing a single halo width could mean some data are communicated unnecessarily. The exchange at line 40 (Example 5-2) is an example of such inefficiency. The stencil of the computations in loop 200 is still one wide in the *i* direction. However, since the halo width of *f* is now 2 in this dimension, one extra halo point on each side for each *j* will be communicated unnecessarily. This extra communication can be eliminated by using a variant of the EXCHANGE directive that only exchanges part of the halo region:

```
CSMS$EXCHANGE(f<1:1>)
```

This option to EXCHANGE tells SMS to exchange only the first halo point in the upper and lower halo regions.

If we were to modify Example 5-2 to use a two dimensional decomposition, the CREATE_DECOMP directive would look as follows:

```
CSMS$CREATE_DECOMP(DECOMP_IJ, <im,jm>, <2,1>)
```

Now, the maximum stencil width is 2 in the first decomposed dimension and 1 in the second decomposed dimension.

5.1.3 Miscellaneous

For exchanges using static memory models the process-local array sizes specified in the DECLARE_DECOMP directive must be large enough to include the halo regions. In the program fragment below, the halo size is one. Since there is a halo region on each side, the declared local array size is the global size (*im*) divided by the number of processes (4) plus 2 to accommodate the halo regions and plus 1 since 4 does not divide 30 evenly.

```
      program STATIC_MEMORY_EXCHANGE
      implicit none
      integer im
      parameter(im = 30)
      integer jm
      parameter(jm = 5)
CSMS$DECLARE_DECOMP(my_dh, <im/4 + 2 + 1>)
```

A second point about EXCHANGE is that, for both static and dynamic memory models, the number of processes used must be small enough to ensure the size of the interior is greater than the halo width in each decomposed dimension.

Finally, we point out that EXCHANGE automatically implements the synchronization required for the parallel code to produce the correct answer. A process scheduled to receive data from a neighbor will wait until the data have fully arrived before proceeding with the next set of calculations.

5.2 Optimizations

In this section, some optimizations are described that can be employed to reduce the number of exchanges and the amount of data exchanged in a parallel SMS program.

5.2.1 Aggregating Exchanges

The program SLOW below, uses a statically declared one dimensional decomposition (line 10) to distribute the arrays a, b and c which contain adjacent dependencies (lines 44, 45, 52). In this example, a halo thickness of one is defined by CREATE_DECOMP (line 24). After a series of iterations (line 39) a global sum is produced with the REDUCE directive (line 63).

```
 1      program SLOW
 2      implicit none
 3      integer im
 4      parameter(im = 30)
 5      integer jm
 6      parameter(jm = 5)
 7      integer iterations
 8      parameter(iterations = 3)
 9
10  CSMS$DECLARE_DECOMP(my_dh, <im/3 + 2>)
11
12  CSMS$DISTRIBUTE(my_dh, <im>) BEGIN
13        real a(im)
14        real b(im,jm)
15        real c(im,jm)
16  CSMS$DISTRIBUTE END
17
18        real ysum
19
20        integer i
21        integer j
22        integer iter
23
24  CSMS$CREATE_DECOMP(my_dh, <im>, <1>)
25
26        ysum = 0.0
27        b = 0.0
28        c = 0.0
29
30        do j = 1, jm
31
32  CSMS$PARALLEL(my_dh, <i>) BEGIN
33          do i = 1, im
34  CSMS$TO_GLOBAL(<1, i>) BEGIN
35            a(i) = real(3*i + 2 + j)
36  CSMS$TO_GLOBAL END
37          end do
38
39          do iter = 1, iterations
40
41  CSMS$EXCHANGE(a)
42
43            do i = 2, im-1
44              b(i,j) = a(i+1) + a(i-1)
45              c(i,j) = b(i,j) + c(i,j)
46            end do
```

```
47
48  CSMS$EXCHANGE(b)
49  CSMS$EXCHANGE(c)
50
51            do i = 2, im-1
52              a(i) = b(i+1,j) + b(i-1,j) + c(i+1,j) - c(i-1,j)
53            end do
54
55          end do
56
57          do i = 2, im - 1
58            ysum = ysum + a(i)
59          end do
60
61        end do
62
63  CSMS$REDUCE(ysum, SUM)
64
65        print *, 'ysum is ', ysum
66  CSMS$PARALLEL END
67        end
```

Example 5-3 A sub-optimal version of a program parallelized using SMS.

SMS provides the capability to aggregate the exchanges of multiple variables. If lines 48-49 are replaced with

```
CSMS$EXCHANGE(b,c)
```

then SMS will combine the communications of the corresponding halo regions of *b* and *c* as shown in Figure 5-10. Since the number of messages sent is halved, performance on high-latency machines will improve.

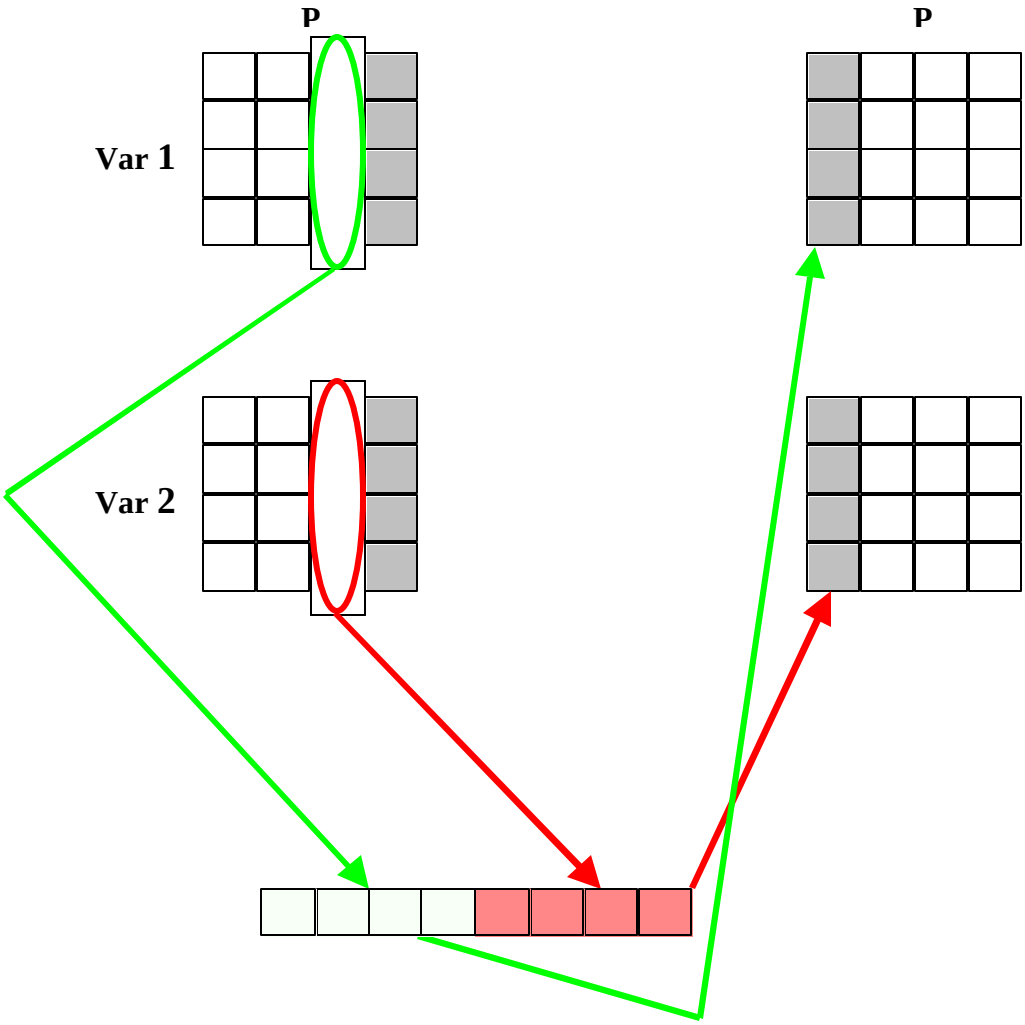


Figure 5-10 An illustration of how communications are aggregated to reduce latency for a portion of the exchange of Var 1 and Var 2. The last column of process P1's variables are communicated as a single message to P2 where they are unpacked into the corresponding halo regions.

5.2.2 Trading Communications for Computations Using HALO_COMP

Example 5-3 can be further optimized by trading communication for redundant computations in the halo region as briefly discussed in the SMS overview paper. This is done using the HALO_COMP directive to modify the ranges of parallel loops to include computations in the halo regions. These extra computations can eliminate the need for some exchanges.

Figure 5-11, Figure 5-12, and Figure 5-13 illustrate how redundant computations work. Without the HALO_COMP directive, *b* and *c* are only computed in interior points using stencils like that shown in Figure 5-11.

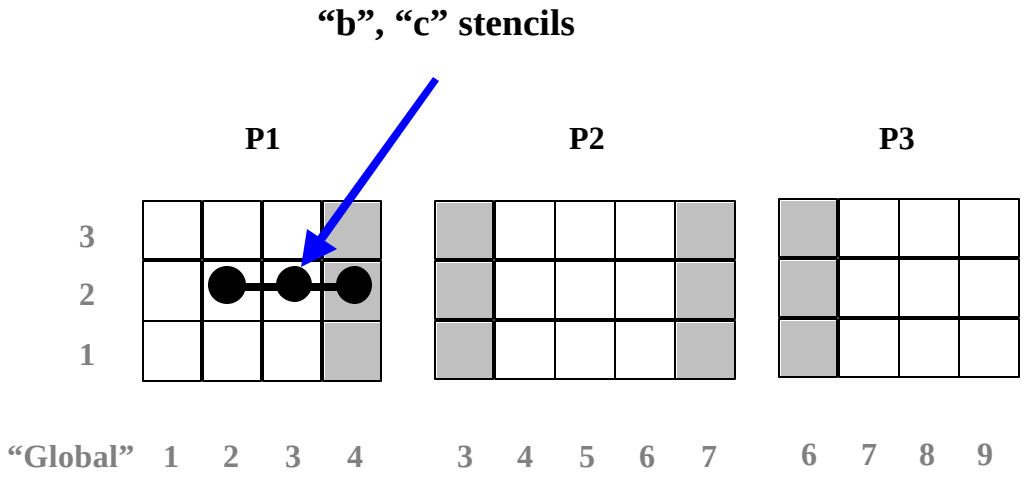


Figure 5-11 Memory layout of "a" (assuming im=9, jm=3) with sample stencil for calculations of "b" and "c" overlaid.

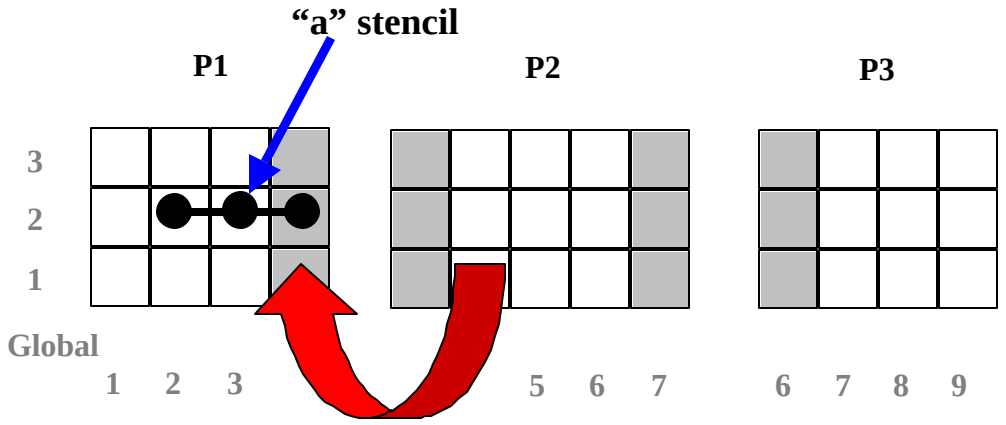


Figure 5-12 Memory layout of "b" and "c" with sample stencil for calculation of "a" overlaid. The halo regions of "b" and "c" must be updated via exchange for the calculation of "a" to be executed correctly.

Halo regions of `b` and `c` must then be updated via an exchange for `a` to be properly computed as shown in Figure 5-12. A computation one step into the halo region (Figure 5-13) requires that `a` have a halo size of two instead of one. Since process P1 now computes points such as `b(4,2)` and `c(4,2)`, the computation of `a(3,2)` shown in Figure 5-12 can proceed without having exchanged `b` and `c`. However, extra computations must be done since process P2 must also perform exactly the same computation for its corresponding points `b(4,2)` and `c(4,2)`,

In this example, the number of exchanges AND the amount of data communicated have been reduced. The amount of data communicated is less because the benefit of not exchanging both *b* and *c* is only partially offset by the fact that the amount of data communicated in the exchange of *a* has doubled.

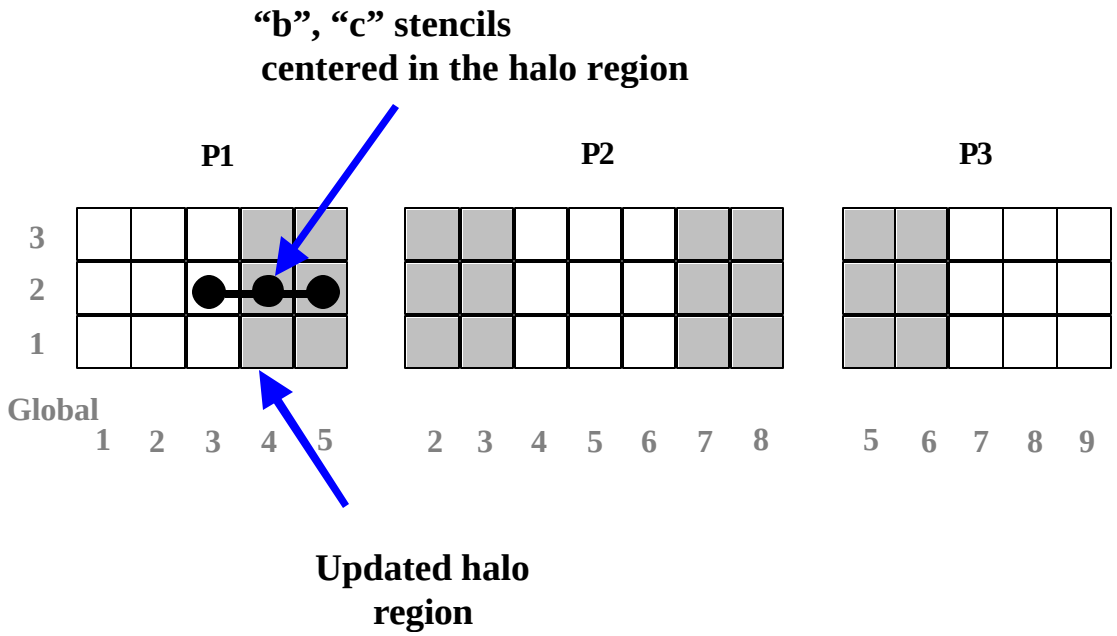


Figure 5-13 Modified memory layout of "a" with new sample stencil centered in the halo region. The computation of point b(4,2) and c(4,2) effectively updates the halo regions of "b" and "c" so that the computation of "a" in Figure 5-12 can be made without an exchange.

A net improvement in performance by this technique will only be realized if the cost of the additional computation by each process is less than the cost of exchanging *b* and *c*. Whether or not the code runs faster will, in general, depend on the communication patterns in the program, the number of processes used, and the target hardware platform. Since adjacent communication does not scale linearly, improved performance will more likely be achieved for a large number of processes on machines where the ratio of communications speed to processor speed is low.

A version of Example 5-3 that implements redundant calculations is shown in Example 5-4. The HALO_COMP directive on line 43 tells SMS that the enclosed loop should be executed 1 step into the halo region in each direction. This updates *b* and *c* sufficiently to satisfy the dependencies in the loop at lines 52-54. DECLARE_DECOMP and CREATE_DECOMP have been modified to accommodate the new halo size of 2. The exchanges of *b* and *c* have been eliminated.

```
 1        program FASTER
 2        implicit none
 3        integer im
 4        parameter(im = 30)
 5        integer jm
 6        parameter(jm = 5)
 7        integer iterations
 8        parameter(iterations = 3)
 9
10  CSMS$DECLARE_DECOMP(my_dh, <im/3 + 4>)
11
12  CSMS$DISTRIBUTE(my_dh, <im>) BEGIN
13        real a(im)
14        real b(im,jm)
15        real c(im,jm)
16  CSMS$DISTRIBUTE END
17
18        real ysum
19
20        integer i
21        integer j
22        integer iter
23
24  CSMS$CREATE_DECOMP(my_dh, <im>, <2>)
25
26        ysum = 0.0
27        b = 0.0
28        c = 0.0
29
30        do j = 1, jm
31
32  CSMS$PARALLEL(my_dh, <i>) BEGIN
33          do i = 1, im
34  CSMS$TO_GLOBAL(<1, i>) BEGIN
35            a(i) = real(3*i + 2 + j)
36  CSMS$TO_GLOBAL END
37          end do
38
39          do iter = 1, iterations
40
41  CSMS$EXCHANGE(a)
42
43  CSMS$HALO_COMP(<1,1>) BEGIN
44            do i = 2, im-1
45              b(i,j) = a(i+1) + a(i-1)
46              c(i,j) = b(i,j) + c(i,j)
47            end do
48  CSMS$HALO_COMP END
49
50
51
52            do i = 2, im-1
53              a(i) = b(i+1,j) + b(i-1,j) + c(i+1,j) - c(i-1,j)
54            end do
```

```
55
56          end do
57
58          do i = 2, im - 1
59            ysum = ysum + a(i)
60          end do
61
62        end do
63
64  CSMS$REDUCE(ysum, SUM)
65
66        print *, 'ysum is ', ysum
67
68  CSMS$PARALLEL END
69
70        end
```

Example 5-4 A version of Example 5-3 that has been optimized by trading communications for redundant calculations in the halo region.

5.2.3 Pulling Exchanges Outside of Loops

Program FASTER is still inefficient on high-latency machines because the exchange of *a* (line 42) occurs inside the *j* loop. To reduce the number of exchanges (thus improving performance) the exchange is moved outside the *j* loop. This requires promoting *a* from a one dimension a two-dimensional array (line 13) and creating a second *j* loop (line 44) as shown in Example 5-5.

```
 1        program FASTEST
 2        implicit none
 3        integer im
 4        parameter(im = 30)
 5        integer jm
 6        parameter(jm = 5)
 7        integer iterations
 8        parameter(iterations = 3)
 9
10  CSMS$DECLARE_DECOMP(my_dh, <im/3 + 4>)
11
12  CSMS$DISTRIBUTE(my_dh, <im>) BEGIN
13        real a(im,jm)
14        real b(im,jm)
15        real c(im,jm)
16  CSMS$DISTRIBUTE END
17
18        real ysum
19
20        integer i
21        integer j
22        integer iter
23
24  CSMS$CREATE_DECOMP(my_dh, <im>, <2>)
25
26        ysum = 0.0
```

```
27        b = 0.0
28        c = 0.0
29
30        do j = 1, jm
31
32  CSMS$PARALLEL(my_dh, <i>) BEGIN
33          do i = 1, im
34  CSMS$TO_GLOBAL(<1, i>) BEGIN
35            a(i,j) = real(3*i + 2 + j)
36  CSMS$TO_GLOBAL END
37          end do
38        end do
39
40        do iter = 1, iterations
41
42  CSMS$EXCHANGE(a)
43
44          do j = 1, jm
45
46  CSMS$HALO_COMP(<1,1>) BEGIN
47            do i = 2, im-1
48              b(i,j) = a(i+1,j) + a(i-1,j)
49              c(i,j) = b(i  ,j) + c(i  ,j)
50            end do
51  CSMS$HALO_COMP END
52
53            do i = 2, im-1
54              a(i,j) = b(i+1,j) + b(i-1,j) + c(i+1,j) - c(i-1,j)
55            end do
56          end do
57
58        end do
59
60        do j = 1, jm
61          do i = 2, im - 1
62            ysum = ysum + a(i,j)
63          end do
64
65        end do
66
67  CSMS$REDUCE(ysum, SUM)
68
69        print *, 'ysum is ', ysum
70
71  CSMS$PARALLEL END
72
73        end
```

Example 5-5 A version of Example 5-3 that has been further optimized by modifying some of the serial code. Array *a* has been promoted to two dimensions to allow the exchange to be placed outside of the *j* loop.

The amount of data communicated by each process (roughly 2**jm*iterations) is unchanged. However, the number of communications is reduced from 2**jm*iterations to 2*iterations. The performance gain from this optimization can be quite dramatic on high latency machines. The

drawbacks of the optimization in this particular case are the increased memory usage (caused by the promotion of *a*) and the slightly increased code complexity.

5.2.4 Using HALO_COMP and TO_LOCAL To Make Subroutines Do Redundant Computations

We saw in Section 4.3 how the TO_LOCAL directive can be used to parallelize subroutines without requiring directives inside the subroutine code. The approach works by making the subroutines operate on the interior of the process local arrays. Now, suppose we want those called routines to do redundant computations in the halo region to avoid communication. Example 5-6 shows a modified version of subroutine *compute* from Example 4-4, illustrating how this is done.

```
 1        subroutine compute
 2        implicit none
 3        include 'tran_index.inc'
 4        integer i, j
 5        integer istart, iend, jstart, jend
 6  CSMS$DISTRIBUTE(DECOMP_IJ, <im>, <jm>) BEGIN
 7        integer x(im,jm), y(im,jm)
 8  CSMS$DISTRIBUTE END
 9
10  CSMS$PARALLEL(DECOMP_IJ,<i>,<j>) BEGIN
11
12  csms$halo_comp(<1,1>, <1,1>) begin
13        do 100 j=1,jm
14        do 100 i=1,im
15  CSMS$TO_GLOBAL(<1,i>, <2,j>) BEGIN
16          x(i,j) = (100 * i) + j
17  CSMS$TO_GLOBAL END
18    100 continue
19  csms$halo_comp end
20
21        y = 0.0
22
23        istart = 1
24        iend   = im - 1
25        jstart = 2
26        jend   = jm
27
28  csms$to_local(<1, im     : size  >, <2, jm   : size  >,
29  csms$>        <1, istart : lbound>, <1, iend : ubound>,
30  csms$>        <2, jstart : lbound>, <2, jend : ubound>) begin
31
32  csms$halo_comp(<1,1>, <1,1>) begin
33
34        call physics(x, im, jm, istart, iend, jstart, jend, y)
35
36  csms$halo_comp end
37
```

```
38    csms$to_local end
39
40          do j = 2, jm
41            do i = 1, im - 1
42              x(i,j) = y(i,j) + y(i+1,j-1)
43            end do
44          end do
45
46    CSMS$SERIAL BEGIN
47          do j = 1, jm
48            write(*,'(16i5)') (x(i,j),i=1,im)
49          end do
50    CSMS$SERIAL END
51
52    CSMS$PARALLEL END
53          return
54          end
```

Example 5-6 Modified version of Example 4-4 that passes lower and upper bounds into subroutine *physics* so that it does redundant computations for one point in the halo region for each dimension and for each direction.

Since the call to *physics* is now contained within both a TO_LOCAL and HALO_COMP directive, the effect is to change the lower and upper bounds passed to the subroutine so that it will do redundant computations for one point in the halo region for each direction, for each decomposed dimension. Figure 5-14 shows the new table of lower and upper bounds (compare to the table in Figure 4-1). Now, following the call to *physics*, variable *y* contains valid data one point into the halo region. Consequently, the loop at lines 40-44 produces the correct answer.

Processor	Decomposed dimension	Size	Lbound	Ubound
P1	1	6	1	5
P1	2	5	1	4
P2	1	6	2	6
P2	2	5	1	4
P3	1	6	1	5
P3	2	5	2	5
P4	1	6	2	6
P4	2	5	2	5

Figure 5-14 Table of sizes, lower bounds and upper bounds for Example 5-6. Compare the lower bounds and upper bounds to the values in the table in Figure 4-1. The sizes are unchanged.

5.3 Debugging Adjacent Dependencies: CHECK_HALO

The analysis of adjacent dependencies in a serial code and the process of accurately placing EXCHANGE and HALO_COMP directives are highly prone to error. To help the user track down such errors, the CHECK_HALO directive and associated SMS_CHECK_HALO environment variable can be used to check if all or part of a halo variable is up-to-date. Suppose, in Example 5-4, the user forgot to include the HALO_COMP directives on lines 43 and 48. When the program is run, it does not produce the correct answer for *ysum*. The user can observe that the loop on lines 52-54 requires one point of the lower and upper halo regions of *b* and *c* up-to-date. To check this assumption, the following directive can be added at line 51:

```
CSMS$CHECK_HALO(b<1:1>, c<1:1>, 'LOOP 52')
```

If the SMS_CHECK_HALO is set to "ON", the generated code checks if the afore-mentioned halo points are up-to-date. In this case, since the halo regions are not up-to-date, the SMS program will generate the following error message and terminate:

```
LOOP 52 Halo check failed for var : b
```

Suppose the HALO_COMP directives are included as shown on lines 43 and 48. This time the check passes so no error messages are generated and the program continues. Suppose the user includes the HALO_COMP directives on lines 43 and 48 and specifies the CHECK_HALO directive as follows:

```
CSMS$CHECK_HALO(b, c, 'LOOP 52')
```

This form of the directive tells SMS to check the entire halo region. Since, for the lower and upper halo regions, only one of the halo points are up-to-date, the program will terminate with the same error message.

The directive can be added to the code on a permanent basis. When SMS_CHECK_HALO is “ON”, CHECK_HALO adds costly communication. However, if the SMS_CHECK_HALO environment variable is set to something other than "ON" then the halo checks are skipped; maximizing performance. If, after a code change, the program generates the wrong answer, the halo checks can be turned back on to help isolate the problem.

6 Handling Complex Dependencies using TRANSFER

Section 2.7 introduced the TRANSFER directive and explained how it could be used to handle complex dependencies in more than one dimension. In Section 6.1, we show how TRANSFER can be used when either the source or destination array are non-decomposed. In Section 6.2, we examine how TRANSFER can be applied to the parallelization of spectral NWP models.

6.1 Further Details about TRANSFER

While TRANSFER can be used to generate communications to transpose two arrays decomposed in one or more dimensions, it can also be used when either the source or destination arrays are not decomposed. If the destination array is not decomposed but the source is then the TRANSFER directive effectively implements a "gather" of the source into the destination as shown in Figure 6-1. After the transfer, the entire array is replicated on each process. Since the local data for each process must be communicated to all other processes, this operation can be quite expensive.

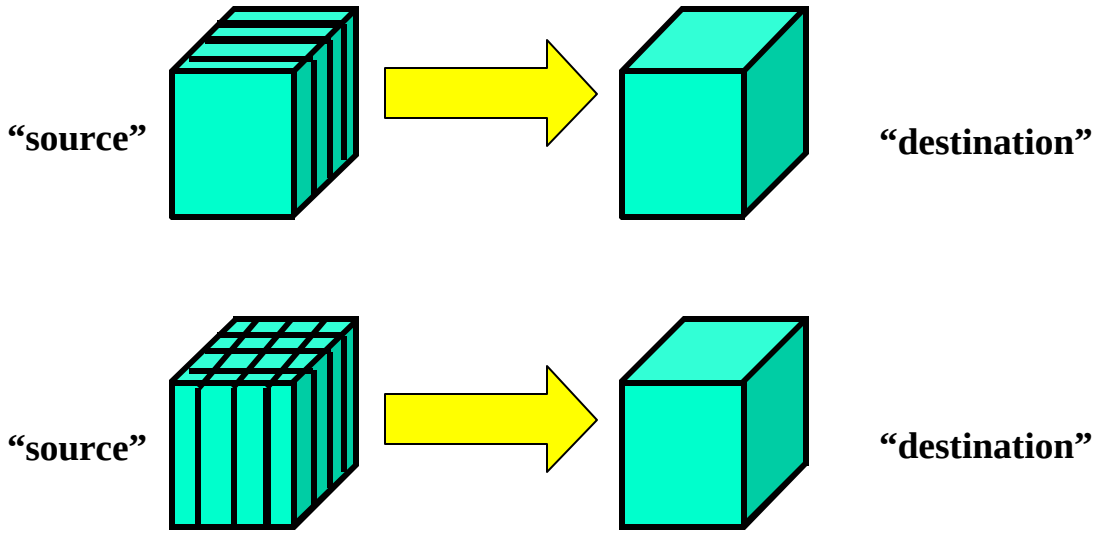


Figure 6-1 Schematic of the behavior of TRANSFER when the source array is decomposed and the destination array is NOT decomposed. The effect is to "gather" the process-local data from the source array into the globally sized destination array.

If the source array is not decomposed but the destination array is decomposed then TRANSFER performs an "extract" of data from the source into the destination as shown in Figure 6-2. Note that no communication is needed in this case since each process has access to all needed data to begin with.

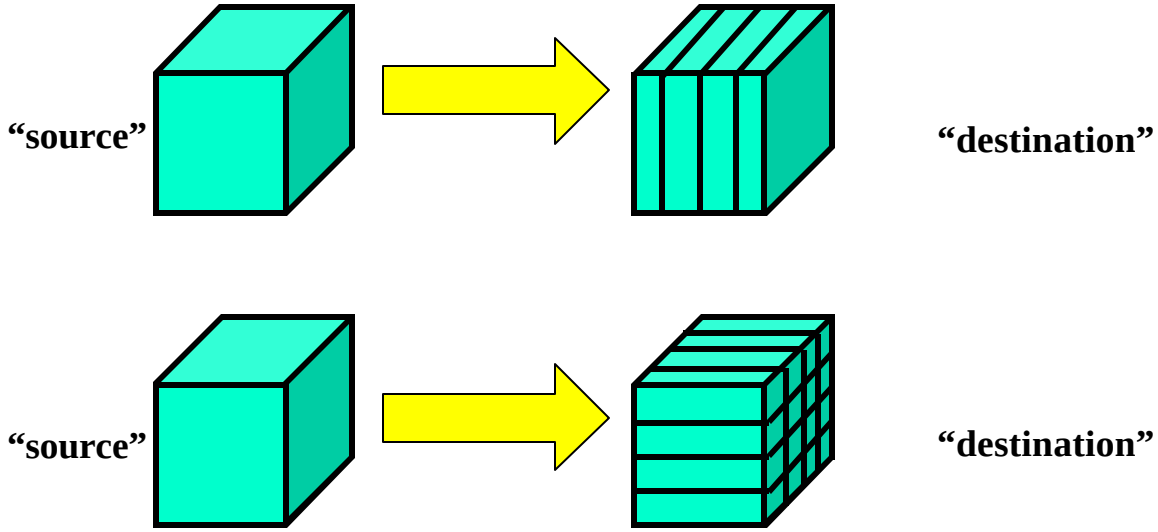


Figure 6-2 Schematic of the behavior of TRANSFER when the source array is NOT decomposed and the destination array is decomposed. The effect is to “extract” the appropriate data from the globally sized source array into the process-local destination array.

As in the case of EXCHANGE, TRANSFERs can be aggregated as follows to reduce latency:

```
CSMS$TRANSFER(<source1, destination1>, <source2, destination2>) BEGIN
```

Serial code here

```
CSMS$TRANSFER END
```

Some dependencies make decomposition in any dimension difficult. The program in Example 6-1 below shows how TRANSFER can be used to avoid parallelization of such code. The idea is to use TRANSFER to gather the data into global arrays (line 28), execute the complex code on the global data (line 35), and then extract from the global data the correct process-local pieces (line 38).

```
 1        program TRANSFER2
 2        implicit none
 3
 4        integer im
 5        parameter(im=60)
 6
 7        integer jm
 8        parameter(jm=90)
 9
10        integer km
11        parameter(km=5)
12
```

```
13  CSMS$DECLARE_DECOMP(DECOMP_IJ, <im/2, jm/2>)
14
15  CSMS$DISTRIBUTE(DECOMP_IJ, im) BEGIN
16        real u(km, im,jm)
17  CSMS$DISTRIBUTE END
18
19  CSMS$INSERT       real u_global(km,im,jm)
20
21  C BEGIN
22
23  CSMS$CREATE_DECOMP(DECOMP_IJ, <im,jm>, <0,0>)
24
25        call manageable_dependencies(u)
26
27  C This is a "gather".
28  CSMS$TRANSFER(<u, u_global>)
29
30  C parallelize later, maybe
31  CSMS$REMOVE BEGIN
32        call nasty_dependencies(u)
33  CSMS$REMOVE END
34
35  CSMS$INSERT       call nasty_dependencies(u_global)
36
37  C This is an "extract".
38  CSMS$TRANSFER(<u_global, u>)
39
40        call more_manageable_dependencies(u)
41
42        end
```

Example 6-1 Example of how TRANSFER can be used to avoid parallelization of code containing complex dependencies.

Notice this variation of the TRANSFER syntax does not have a BEGIN and END directive (no serial code is replaced in this case). This example illustrates how SMS can be used to parallelize a program in pieces while still producing the correct answer. If the subroutine *nasty_dependencies* consumes a small amount of serial run-time and the parallel code need only scale to a few processes then the modeler may choose never to parallelize this routine. The INSERT and REMOVE directives are used to replace the serial code that references *u* with code that references *u_global*. These directives will be explained in Section 8.2. Section 8.1 will show how to avoid this parallelization even more easily using the SERIAL directive, although possibly at the cost of performance.

6.2 Applying TRANSFER to Spectral NWP Models

Many spectral NWP models have multiple phases of computation that repeat in a fixed pattern. Phases often have different optimal decompositions, so performance may be maximized by using

multiple decompositions and transferring between them. Consider the case of one dimensional decompositions for these models. The physical parameterizations contain complex dependencies in the vertical. This makes it efficient to decompose in one of the horizontal dimensions. At the same time, many computer system vendors provide highly optimized assembly FFT libraries that far out-perform anything that can be done with hand-tuned Fortran code. Taking advantage of this serial code requires decomposing in a dimension other than `i`. So, typically, the data are decomposed in the `j` dimension during physics and FFT computations. This is decomposition "a" already seen in Figure 3-1. The Legendre transformations contain complex dependencies in the `j` dimension. Therefore, a second decomposition in either `i` (decomposition "b" in Figure 3-1) or `k` (decomposition "c" in Figure 3-1) is needed for optimal performance during these calculations. The TRANSFER directive provides the means to transpose the data from decomposition "a" to ("b" or "c") and back again.

A future release of this users guide will include an example illustrating how TRANSFER can be used to help parallelize a simple spectral code.

7 Handling Global Dependencies Using REDUCE

In Section 2.3, we saw how the REDUCE directive was used to implement communication needed to do global summations and maxima. In this section we examine other forms of the REDUCE directive. In addition to global summations and maxima, the REDUCE directive can be used to generate global minima. Reductions of arrays are also supported. Section 7.1 discusses these points. As we will see, the form of REDUCE introduced in Section 2.3 (which will be referred to as "Standard Reductions") does not necessarily produce the bit-wise exact same answer as the serial code for global summations of floating point numbers. Section 7.2 introduces a second form of REDUCE called "Bit-wise Exact" that does produce the bit-wise same answer, regardless of the number of processes.

7.1 More on Standard Reductions

Example 7-1 shows additional examples of standard reductions. Global minima are generated by specifying the keyword ***MIN*** (line 52). Also notice that reductions can be aggregated in the same way as exchanges (line 50). One of the variables reduced is the non-decomposed array *xsum* (line 50). The summation of *xsum* looks like the following:

```
Xsum_global(1) = Xsum_local1 (1) + Xsum_local2 (1) + ...
Xsum_global(2) = Xsum_local1 (2) + Xsum_local2 (2) + ...
     .
     .
     .
```

where *Xsum_local$_P$(j)* is the value of process-local *xsum(j)* on process P and *Xsum_global* is the value of *xsum* after the global summation is complete.

```
 1        program REDUCTIONS
 2        implicit none
 3        include 'basic.inc'
 4
 5        im = 50
 6        jm = 2
 7
 8  CSMS$CREATE_DECOMP(DECOMP_I, <im>, <0>)
 9
10        call DO_THEM
11
12        end
13
14        subroutine DO_THEM
15        implicit none
16        include 'basic.inc'
17
18  CSMS$DISTRIBUTE(DECOMP_I, <im>) BEGIN
19        real x(im,jm)
```

```
20        real y(im,jm)
21    CSMS$DISTRIBUTE END
22
23        real xsum(jm)
24        real ysum
25        real xmin
26        real xmax
27
28        integer i
29        integer j
30
31        open (10, file='reduce_data', form='unformatted')
32        read (10) x, y
33        close(10)
34
35    CSMS$PARALLEL(DECOMP_I, <i>) BEGIN
36        xsum = 0.0
37        ysum = 0.0
38        xmax = x(1,1)
39        xmin = x(1,1)
40
41        do j = 1, jm
42          do i = 1, im
43            xsum(j) = xsum(j) + x(i,j)
44            ysum = ysum + y(i,j)
45            xmax = max(xmax, x(i,j))
46            xmin = min(xmin, x(i,j))
47          end do
48        end do
49
50    CSMS$REDUCE(xsum, ysum, SUM)
51    CSMS$REDUCE(xmax,       MAX)
52    CSMS$REDUCE(xmin,       MIN)
53
54        print *
55        print *, 'Global values'
56        do j = 1, jm
57          write(*,100) j, xsum(j)
58        end do
59        write(*,150) ysum
60        write(*,200) xmax
61        write(*,300) xmin
62
63     100  format('j ', i2, ' xsum = ', F13.5)
64     150  format('ysum = ', F13.5)
65     200  format('xmax = ', F13.5)
66     300  format('xmin = ', F13.5)
67
68    CSMS$PARALLEL END
69
70        return
71        end
```

Example 7-1 Program showing additional examples of how the REDUCE directive can be used.

If we were to modify Example 7-1 so that the *j* dimension is also decomposed then xsum would be a decomposed variable. In this case, the reduction of *xsum* would FAIL because SMS does not currently support reductions that produce decomposed variables. This would require doing the reduction over a subset of the processes. Support for such reductions will be added in a future SMS release.

When run with 2 processes, program REDUCTIONS yields the following results:

```
 Global values
j  1 xsum =    1258.28589
j  2 xsum =    1310.71448
ysum =   -2464.28540
xmax =     100.00000
xmin =    -100.00000
```

However, when run with 4 processes, the results are :

```
 Global values
j  1 xsum =    1258.28577
j  2 xsum =    1310.71436
ysum =   -2464.28613
xmax =     100.00000
xmin =    -100.00000
```

Notice that the values for *xsum* and *ysum* are slightly different between the 2 and 4 process runs. We will now see why this is the case.

7.2 Bit-wise Exact Reductions

The differences in results in Example 7-1 are due to round-off error in the floating point addition. The numbers are added in a different order in the 4 process case as compared to the 2 process case because, as discussed in Section 2.3.3, the sums are first computed locally before being combined. In NWP models (which are non-linear systems), if the global sums feed back into the main model equations, these slight errors can grow and propagate; potentially yielding completely different model predictions for runs with differing numbers of processes.

For testing purposes, it is useful to avoid these round-off errors. To do this, SMS offers a form of REDUCE that produces the bit-wise exact same answer for any number of processes. Example 7-2 below shows how this works.

```
1        program EXACT_REDUCTIONS
2        implicit none
3        include 'basic.inc'
4
5        im = 50
6        jm = 2
7
8   CSMS$CREATE_DECOMP(DECOMP_I, <im>, <0>)
```

```
 9
10        call DO_THEM
11
12        end
13
14        subroutine DO_THEM
15        implicit none
16        include 'basic.inc'
17
18  CSMS$DISTRIBUTE(DECOMP_I, <im>) BEGIN
19        real x(im,jm), y(im,jm)
20  CSMS$DISTRIBUTE END
21
22        real ysum
23
24        integer i
25        integer j
26
27        open (10, file='reduce_data', form='unformatted')
28        read (10) x, y
29        close(10)
30
31  CSMS$PARALLEL(DECOMP_I, <i>) BEGIN
32
33  CSMS$REDUCE(<y, ysum>, SUM) BEGIN
34        ysum = 0.0
35        do j = 1, jm
36          do i = 1, im
37            ysum = ysum + y(i,j)
38          end do
39        end do
40  CSMS$REDUCE END
41
42        print *
43        print *, 'Global values'
44        write(*,150) ysum
45
46   150  format('ysum = ', F13.5)
47
48  CSMS$PARALLEL END
49
50        return
51        end
```

Example 7-2 Program illustrating the bit-wise exact form of the REDUCE directive.

The modified REDUCE syntax can be see on lines 33 and 40. The syntax requires a BEGIN and END directive. The BEGIN directive lists the variable being summed (*y*) and the resulting global sum (*ysum*). The keyword ***SUM*** is also included but is, in general, optional since this version of REDUCE only supports global sums. The serial code between the two directives MUST sum *y* and store the result in *ysum*. SMS replaces these calculations with code that gathers each process's piece of *y* into a globally-sized (replicated) variable and then sums the

result in the correct order. Conceptually, the generated parallel code would look like the following:

```
call GATHER(y, y_global)
ysum = 0.0
do j = 1, jm_global
  do i = 1, im_global
    ysum = ysum + y_global(i,j)
  end do
end do
```

The "gather" operation is done in the same way as TRANSFER was used to gather variables as discussed in Section 6.1. Since the gather operation requires communicating the entire contents of *y* to all processes, this form of global sum is significantly less efficient than the "Standard" form. In that case, only the process-local scalar sums were communicated to all the processes.

Even in the bit-wise exact form, the REDUCE directive will only produce bit-wise exact sums if an environment variable called SMS_BITWISE is set to the value EXACT. Running EXACT_REDUCE in a c-shell environment might look as follows:

```
>> setenv SMS_BITWISE EXACT
>> smsRun 2 exact_reduce
SMS:  BITWISE EXACT reductions will be used when requested.
 Global values
ysum =    -2464.28418
```

Notice that the message printed by SMS regarding reductions now indicates that bit-wise exact reductions will be used.

If SMS_BITWISE is NOT set to EXACT then the effect of the REDUCE directive is the same as in the "standard" reduction; each process computes a local sum of *y* and the resulting scalars are summed across the processes.

```
>> setenv SMS_BITWISE INEXACT
>> smsRun 2 exact_reduce
SMS:  Standard reductions will be used.
 Global values
ysum =    -2464.28540
```

Notice that the answer is the same as that seen in Example 7-1 for the 2 process case.

An important subtle point to make about the bit-wise exact syntax is that the REDUCE BEGIN/END directives and enclosed code MUST be contained within a PARALLEL region. Otherwise, in Example 7-2, when SMS_BITWISE is NOT set to EXACT, the global versions of the loops starting at line 35 would execute even though *y* is decomposed; generating an out-of-bounds error. In actuality, SMS detects this mistake and generates the following syntax error message:

```
Bit-wise exact reductions must be in a parallel region.
```

In summary, the "bit-wise exact" form of global summation is valuable for testing purposes, particularly for non-linear systems. However, for long model runs, when optimal performance is important, the "standard" form of REDUCE will likely be more appropriate because it is much faster. The programmer can use the bit-wise exact form of REDUCE in the code and then decide at run-time, with the SMS_BITWISE environment variable, which reduction to use.

8 Other Directives

There will be instances where the SMS directives seen so far are not sufficient to parallelize a section of code. Several directives are introduced to handle these cases. They are: SERIAL, INSERT, REMOVE, and IGNORE. These are usually the directives of last resort.

8.1 SERIAL

Many cases where the previously discussed SMS directives can not be easily applied to a piece of serial code occur in portions of models where efficient performance is not critical. One example is initialization. For long model runs, the effects of inefficient code during initialization become negligible. Diagnostic print messages are another case. If the user can turn off diagnostic messages when high performance is needed then the presence of inefficient parallel code that generates these messages does not pose a problem. We saw a third case in Example 6-1 where it may be acceptable to leave a piece of the original code un-parallelized because its computations represent only a small fraction of the total run-time of the program.

The SERIAL directive is the easiest way to generate code that produces the right answer in these cases. The directive defines a region over which serial computations will be done. The directive looks as follows:

```
CSMS$SERIAL BEGIN

    Code to run serially

CSMS$SERIAL END
```

Fundamentally, the code contained between the SERIAL BEGIN and END directives is executed by one processor; just as if the code were being run serially instead of as part of a parallel SMS program. For the code to produce the correct answer, it must operate on global, not decomposed arrays. Therefore, any decomposed arrays referenced within the serial region must be gathered into global equivalents before the designated processor executes the code. After the code is executed, any of these gathered global arrays that are modified must be scattered back to all the processes. In addition, any non-decomposed variables that have been modified must be broadcast to all the processes. Since determining what data have been modified is non-trivial, particularly in the case where they are modified via subroutine call, SMS currently gathers/scatters all decomposed variables and broadcasts all non-decomposed variables referenced in the code between the SERIAL BEGIN/END. This communication causes the code to run even more slowly than the original serial code.

In Example 8-1, `x` and `y` are decomposed while `z` is not. The subroutine calls at lines 39-40 read in `x` and `z` using C language routines. These routines cannot be handled by SMS. The print statement at line 41 could be handled by using TRANSFER to gather `y` into a global variable (call it `y_global`) and then printing `y_global`(2,2). However, application of the

SERIAL directive is simpler. PPP generates code that gathers x and y into global variables. A designated processor then executes the code at lines 39-41. Finally, the generated code scatters x and y and broadcasts the value of z. When high performance is desired, the user can avoid this poorly performing code by setting ENABLE_DIAGS to .false.

```
[Include file: serial.inc]
      1      integer im,jm
      2      common /sizes_com/ im,jm
      3 CSMS$DECLARE_DECOMP(DECOMP_IJ)

[Source file: serial1.f]
      1       program SERIAL
      2
      3       include 'serial.inc'
      4
      5       integer i
      6       integer j
      7
      8       im = 5
      9       jm = 4
     10
     11 CSMS$CREATE_DECOMP(DECOMP_IJ, <im,jm>, <0,0>)
     12
     13       call DO_IT
     14
     15       end
     16
     17       subroutine DO_IT
     18       include 'serial.inc'
     19
     20 CSMS$DISTRIBUTE(DECOMP_IJ, <im>, <jm>) BEGIN
     21       real x(im,jm)
     22       real y(im,jm)
     23 CSMS$DISTRIBUTE END
     24       real z
     25       logical ENABLE_DIAGS
     26       ENABLE_DIAGS = .true.
     27
     28       open(10, file='yin', form='unformatted')
     29       read(10) y
     30       close(10)
     31
     32 C Some parallel computations
     33 C          .
     34 C          .
     35 C          .
     36
     37       if (ENABLE_DIAGS) then
     38 CSMS$SERIAL BEGIN
     39         call READ_2D_ARRAY_USING_C(x, im, jm)
     40         call READ_SCALAR_USING_C(z)
     41         print *, 'y(2,2), z ', y(2,2), z
     42 CSMS$SERIAL END
```

```
43       end if
44 C More parallel calculations
45            .
46            .
47       return
48       end
```

Example 8-1 A sample program showing how the SERIAL directive can be used to generate correct parallel code in a simple fashion when other SMS directives will not suffice.

Example 8-2 shows a modified version of Example 6-1 that uses the SERIAL directive. The solution that uses the SERIAL directive is much simpler.

```
 1       program TRANSFER3
 2       implicit none
 3
 4       integer im
 5       parameter(im=60)
 6
 7       integer jm
 8       parameter(jm=90)
 9
10       integer km
11       parameter(km=5)
12
13 CSMS$DECLARE_DECOMP(DECOMP_IJ, <im/2, jm/2>)
14
15 CSMS$DISTRIBUTE(DECOMP_IJ, im) BEGIN
16       real u(km, im,jm)
17 CSMS$DISTRIBUTE END
18
19
20 C BEGIN
21
22 CSMS$CREATE_DECOMP(DECOMP_IJ, <im,jm>, <0,0>)
23
24       call manageable_dependencies(u)
25
26 C parallelize later, maybe
27 CSMS$SERIAL BEGIN
28       call nasty_dependencies(u)
29 CSMS$SERIAL END
30
31       call more_manageable_dependencies(u)
32
33       end
```

Example 8-2. Simpler version of Example 6-1 using the SERIAL directive.

Although useful, the serial directive has some important restrictions. One is that subroutines called from within a serial region may not modify common block variables unless they are passed as arguments. So suppose in Example 8-1, we insert

```
call sub1
```

after line 38. Further suppose *sub1* looks as follows:

```
subroutine sub1
real xc
common /com1/ xc
xc = 2.0
return
end
```

PPP has no way of knowing that *xc* has to be broadcast before the end of the serial region because it does no inter-procedural analysis. If *xc* were an argument passed to *sub1* then the SERIAL directive could be used.

A second case where a SERIAL directive cannot be used is shown in Example 8-3. Here, the constant 2 is passed to subroutine *DO_IT*. Since *DO_IT* calls a C routine that uses dummy argument *n*, a SERIAL directive would normally be required to handle this. However the SERIAL directive generates a broadcast of dummy argument *n*. This broadcast will attempt to write to variable *n*. Since variable *n* is the constant 2, the result will be, at best, a core dump. The solution would be to assign 2 to a variable in the main program and pass the variable to subroutine *DO_IT*.

```
 1       program SERIAL
 2
 3       include 'serial.inc'
 4
 5       integer i
 6       integer j
 7
 8       im = 5
 9       jm = 4
10
11 CSMS$CREATE_DECOMP(DECOMP_IJ, <im,jm>, <0,0>)
12
13       call DO_IT(2)
14
15       end
16
17       subroutine DO_IT(n)
18
19       integer n
20
21 CSMS$SERIAL BEGIN
22       call c_routine(n)
```

```
23 CSMS$SERIAL END
24
25       return
26       end
```

Example 8-3 Example code where use of the SERIAL directive generates parallel code that fails to run properly.

8.2 INSERT and REMOVE

Two directives, INSERT and REMOVE, are used to modify source code directly. Working together, these directives are very useful for translating code that cannot be converted using other SMS directives. Each line that the user wishes to insert must be prefaced by INSERT. The inserted code that follows must adhere to Fortran 77 fixed format rules. REMOVE removes all text between the directive's BEGIN and END statements.

```
CSMS$REMOVE BEGIN
Code that will not be executed in the SMS program
CSMS$REMOVE END
CSMS$INSERT      Code that will be executed in the SMS program
```

Example 6-1 showed how these directives can be used.

8.3 IGNORE

IGNORE is another directive used to manipulate code directly. This directive instructs PPP to ignore any text between the directive's BEGIN and END. This allows the user to prevent modifications of the serial code by PPP.

```
CSMS$PARALLEL(dh,<i>,<j>) BEGIN
      do 200 i=1, nx
        do 200 j=1, ny
          z(i,j,k) = z(i,j,k)+ y(i,j,k)
200   continue
CSMS$IGNORE BEGIN
      do 300 i=1,3
  call smooth(z)
300   continue
CSMS$IGNORE END
CSMS$PARALLEL END
```

Example 8-4. Using IGNORE to prevent PPP translation.

In Example 8-4, the enclosing parallel region around the 200 loop will ensure translation of the loop variable “i” to a local value. However, we do not wish to translate the 300 loop because “i” is used to iterate on the function “smooth”. To avoid translation of this loop, the IGNORE is used. Optionally, the parallel region could be ended before the 300 loop and then started again after the iteration loop ends.

9 I/O

One of the most powerful features of SMS is its ability to support most types of I/O without requiring any directives. In particular, this is the case for unformatted I/O of scalars and complete arrays, as will be discussed in Section 9.1. The fact that communication patterns for I/O of decomposed and non-decomposed arrays differ is hidden from the programmer. In either case, SMS automatically generates the communication needed to read or write data to or from disk in the same sequence as the serial code would have done it, regardless of the number processes used. By default, SMS assumes the data are stored in native Fortran binary format on disk. However, SMS provides environment variables that can be set to change this default as discussed in Section 9.1.

The I/O of pieces of arrays do require special attention as will be discussed in Section 9.2. Formatted input is, for the most part, handled automatically. However, there are some limitations that will be described in Section 9.3. As discussed in previous sections, formatted output sometimes requires the programmer to specify if and how the data should be printed. SMS allows the user to make these decisions by providing several print modes as will also be discussed in 9.3. Finally, SMS offers several easy-to-use methods for improving I/O performance as discussed in Section 9.4.

9.1 General Unformatted I/O

Figure 9-1 illustrates dependencies for read and write of a simple one-dimensional decomposed array. During a read, data from a single file must be parceled out to each process. This type of communication pattern is called "scatter". During a write, data from each process must be combined in the proper order and written to disk. This type of communication pattern is a different form of "gather" than that seen for TRANSFER and bit-wise exact REDUCE. In this case, instead of gathering the data into a global variable that is replicated in memory on all processes, it is gathered into a single file on disk. "Proper order" means the data must be read from or written to disk in the same sequence as the serial code would have done it. Though it appears quite simple in Figure 9-1, the data reorganization required to match serial ordering in files can be quite complex, especially for two-dimensional decompositions or when the decomposed arrays have halo regions (Figure 9-2). Additionally, when variables being input have halo regions associated with them, these regions will be automatically updated by SMS.

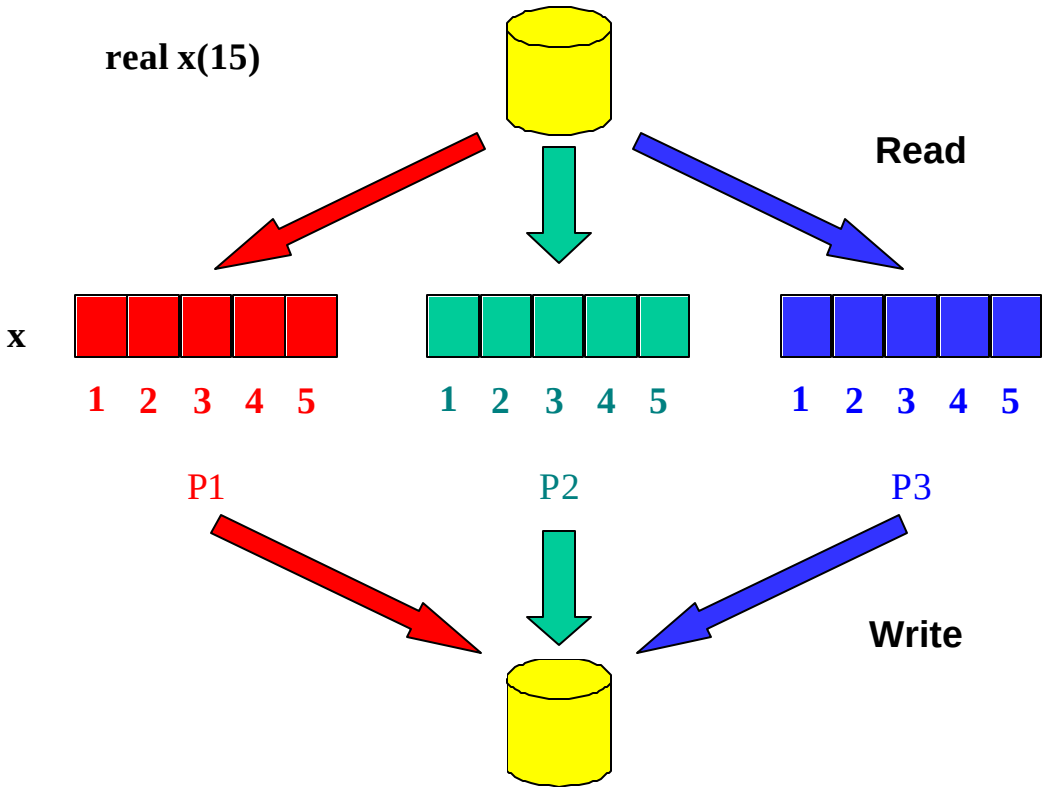


Figure 9-1 Schematic of the input and output of a decomposed array. On input, one process reads the global data from disk. The appropriate sections of the global array are then "scattered" to each process. On output, the decomposed data are gathered into a global array and then written to disk.

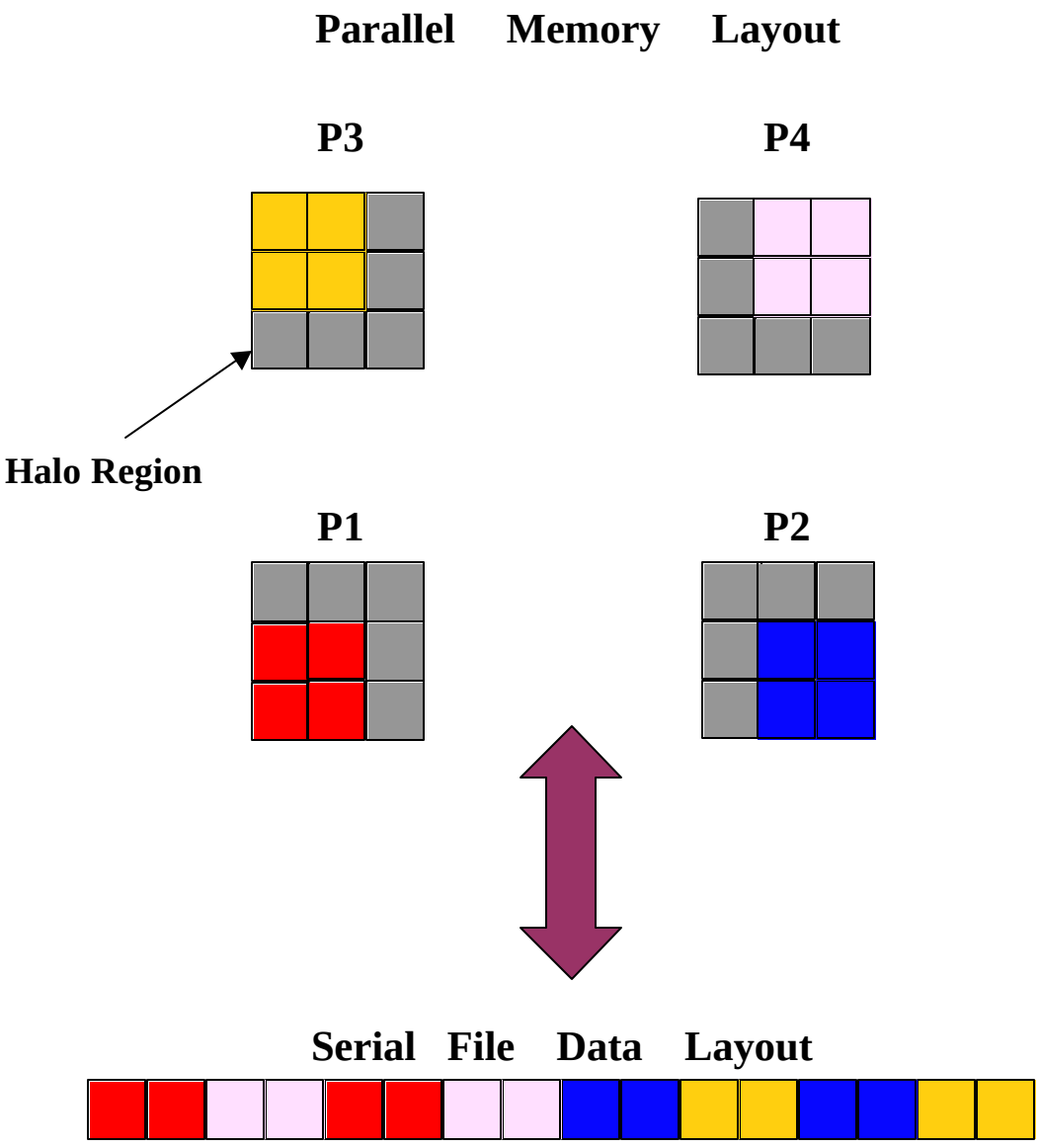


Figure 9-2 Schematic of the re-ordering required to write and read two-dimensionally decomposed data to disk in the same order as the serial code would write it. Special care has to be taken to write the only the interior of each process-local domain and not the halo data. The halo regions are filled during the read operations.

Figure 9-3 illustrates dependencies for read and write of a non-decomposed variable. During a read, a copy of data from a single file must be sent to each process. This type of communication pattern is called "broadcast". During write, it is only necessary to write data from a single process because each process should have an identical copy of the variable.

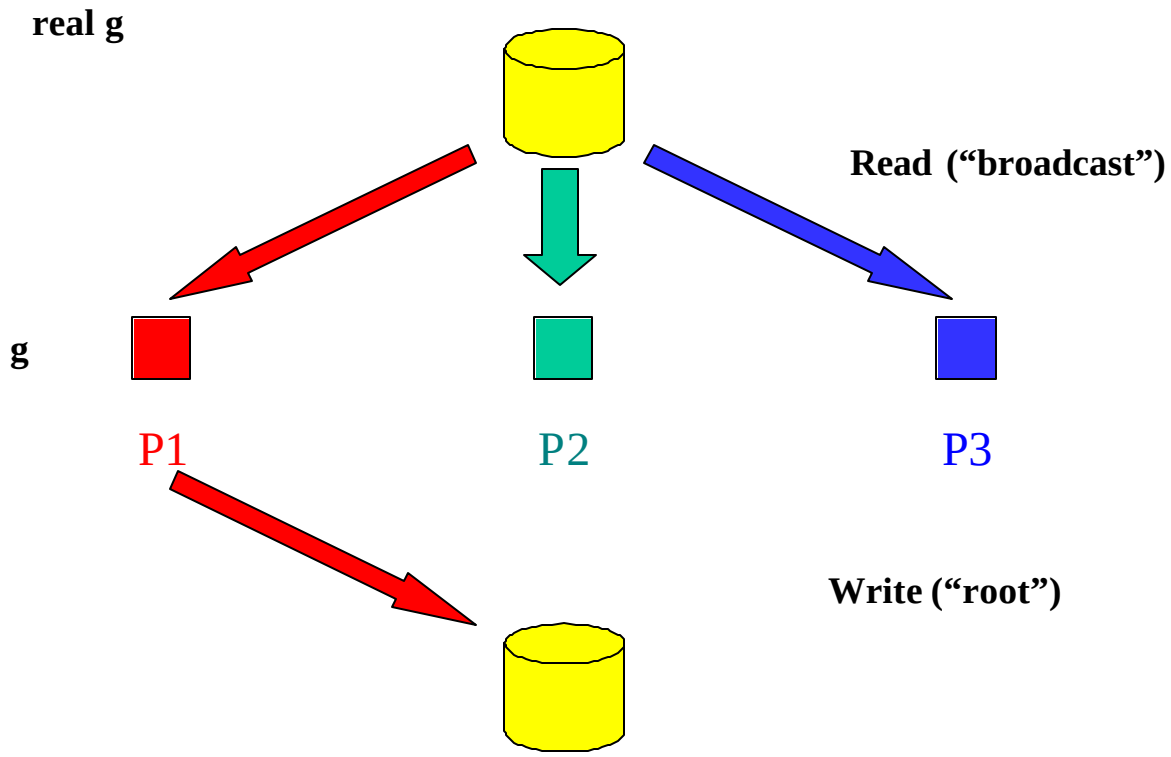


Figure 9-3 Schematic of the input and output of a non-decomposed array. On input, one process reads the data from disk. The data are then replicated on all other processes. On output, a designated “root” process writes the data to disk.

Example 9-1 demonstrates unformatted I/O of both decomposed and non-decomposed variables.

```
[Include file:  io.inc]

 1        integer im, jm
 2        common /sizes_com/ im, jm
 3  CSMS$DECLARE_DECOMP(DECOMP_IJ)
```

[Source file: binary.f]

```
 1        program binary_io
 2        include 'io.inc'
 3        im = 10
 4        jm = 5
 5  CSMS$CREATE_DECOMP(DECOMP_IJ, <im,jm>, <1,0>)
 6        call write_data
 7        end
 8
 9        subroutine write_data
10        include 'io.inc'
11        integer i, j
12        real scale
13  CSMS$DISTRIBUTE(DECOMP_IJ, <im>, <jm>) BEGIN
14        integer x(im,jm), y(im,jm)
15  CSMS$DISTRIBUTE END
16  CSMS$PARALLEL(DECOMP_IJ,<i>,<j>) BEGIN
17        do j=1,jm
18          do i=1,im
19  CSMS$TO_GLOBAL(<1,i>, <2,j>) BEGIN
20            x(i,j) = (100 * i) + j
21            y(i,j) = mod(i,2)
22  CSMS$TO_GLOBAL END
23          end do
24        end do
25  CSMS$PARALLEL END
26        scale = -1.0
27        open (17,file='io1_out.dat',form='unformatted')
28        write (17) x, y, scale
29        close (17)
30
31        open (18,file='io1_out.dat',form='unformatted')
32        read  (18) x, y, scale
33        close (18)
34        return
35        end
```

Example 9-1 Program that does output of both decomposed and non-decomposed data. No additional directives are required for the correct output to be produced, regardless of the number of processes.

In Example 9-1, SMS automatically translates all the read and write statements for both decomposed arrays *x* and *y* and non-decomposed scalar *scale* to the appropriate parallel I/O operations. When automatically generating parallel I/O operations, PPP uses information in the DISTRIBUTE directives to determine how to generate communications to satisfy the I/O dependencies. Notice that any types of decomposed or non-decomposed variables can be mixed in a single write or read statement. It is not necessary to reorganize existing serial read or write statements to take advantage of automatic parallelization by SMS.

By default, SMS assumes unformatted files are stored in native FORTRAN binary format. The default behavior can be modified using the following environment variables:

```
SMS_READ_FORMAT
SMS_WRITE_FORMAT
SMS_IO_FORMAT
```

If the user specifies both SMS_IO_FORMAT and SMS_READ_FORMAT then SMS_READ_FORMAT takes precedence.

If the user specifies both SMS_IO_FORMAT and SMS_READ_FORMAT then the following warning will be printed at the beginning of the run:

```
SMS: Warning!  SMS_IO_FORMAT ignored; SMS_READ_FORMAT takes precedence.
```

The same holds for SMS_WRITE_FORMAT.

The currently available (case insensitive) formats are:

```
IBM
SUN
SGI
FUJITSU
HP
DEC
COMPAQ
IA32
MPI
MPI_EXTERNAL
EXTERNAL
SMS
```

Note that, in many cases, file formats with different names are actually the same format. For example, SGI and SUN are really the same format. It is also important to point out that MPI, MPI_EXTERNAL, EXTERNAL, and SMS all refer to the portable MPI I/O external format. The advantage to using this format is that any file written by an SMS program may be read by any other SMS program on any other machine. This is true regardless of the number of processes used on either machine because SMS preserves serial data ordering.

To convert data files from one format to another, simply write a serial program that reads and writes the data, compile and link with SMS and then set the afore-mentioned environment variables appropriately.

9.2 Unformatted I/O of Elements of Decomposed Arrays.

Some NWP models require I/O of pieces of decomposed arrays. We saw in Section 8.1 how the SERIAL directive could be used to do this. Example 9-2 shows a more efficient solution to this problem.

```
 1       program WRITE_POINTS
 2       include 'io.inc'
 3       im = 10
 4       jm = 5
 5 CSMS$CREATE_DECOMP(DECOMP_IJ, <im,jm>, <1,0>)
 6       call compute
 7       end
 8
 9       subroutine compute
10       include 'io.inc'
11 CSMS$DISTRIBUTE(DECOMP_IJ, <im>, <jm>) BEGIN
12       integer x(im,jm)
13 CSMS$DISTRIBUTE END
14
15       open (10, file='io1_out.dat', form='unformatted')
16       read (10) x
17       close(10)
18       call write_point_data(x)
19       return
20       end
21
22       subroutine write_point_data(x)
23       include 'io.inc'
24
25 CSMS$DISTRIBUTE(DECOMP_IJ, <im>, <jm>) BEGIN
26       integer x(im,jm)
27 CSMS$DISTRIBUTE END
28
29 CSMS$INSERT      integer xpt(2), ipt
30
31 CSMS$INSERT      do ipt=1,2
32 CSMS$INSERT        xpt(ipt) = 0
33 CSMS$INSERT      end do
34
35 CSMS$PARALLEL(DECOMP_IJ) BEGIN
36 CSMS$GLOBAL_INDEX(1,2) BEGIN
37
38 CSMS$INSERT      xpt(1) = x(1,1)
39 CSMS$INSERT      xpt(2) = x(im/2,jm/2)
40
41 CSMS$GLOBAL_INDEX END
42 CSMS$PARALLEL END
43
44 CSMS$REDUCE(xpt,SUM)
45       open (17,file='io2_out.dat',form='unformatted')
46
47 CSMS$REMOVE BEGIN
48       write (17) x(1,1), x(im/2,jm/2)
49 CSMS$REMOVE END
50
51 CSMS$INSERT      write (17) xpt
52
53       close (17)
54       return
```

```
55        end
```

Example 9-2 A program that illustrates how SMS can be used to output pieces of decomposed arrays efficiently.

In Example 9-2, subroutine *write_point_data* outputs two data points of array *x* to unformatted file *io2_out.dat*. Since both dimensions of array *x* are decomposed, it is likely that the two data points will not be on a single process. Other processes may have no data to write. The code at lines 31-33 initializes *xpt* to 0 for every process. The GLOBAL_INDEX directive ensures the code on lines 38-39 assigns to *xpt* the correct values to be written only for the process(es) that contain(s) the correct data points. Finally, the REDUCE directive at line 44 stores in *xpt* the correct answer for every process by summing the zero and non-zero values.

For example, suppose after line 38, xpt(1) looks as follows:

```
Process         1          2         3
Data            0          502       0
```

The REDUCE directive will globally sum *0*, *502* and *0*. The resulting sum (*502*) is stored in *xpt* for every process. Now the write statement on line 51 can write the correct value of *x(1,1)* to disk.

9.3 Formatted I/O

9.3.1 Formatted Input

Formatted input including namelists is handled automatically by SMS. The user does not need to add any directives. The only caveat is that input variables cannot be decomposed arrays. In this case, a work-around is to enclose the formatted read statements within a SERIAL directive. Since formatted reads typically occur infrequently during the course of a model run, this approach usually does not incur a significant performance penalty.

9.3.2 Formatted Output

Formatted output requires further discussion. The simple task of printing a message on the screen becomes more complicated in an SPMD programming model. Consider the following simple print statement:

```
      print *,'HELLO'
```

There are no clear standard definitions of what will appear on the screen when a "parallel" print statement is executed. Will each process print a separate message? Will the separate messages appear on different lines on the screen? Will all processes be forced to wait until the print is complete before useful work can continue? If the statement were executed on three processes, we might see any of the following output:

```
HELLO

HELLO
HELLO
HELLO

HHHEEELLLLLLOOO

HELLHEHLEOLOLLO
```

During the brief history of parallel computing, each of these possibilities has been implemented on at least one parallel machine.

SMS simplifies this situation by providing three "print modes" that allow the user to control the behavior of parallel print. The modes are ROOT, ASYNC, and ORDERED. These print modes are illustrated in the following example and the subsequent discussion. Assume, line 18 of subroutine COMPUTE in Example 9-2 is replaced with:

```
      call print_stat(x)
```

Subroutine print_stat is as follows:

```
 1        subroutine print_stat(x)
 2        include 'io.inc'
 3        integer i, j
 4  CSMS$DISTRIBUTE(DECOMP_IJ, <im>, <jm>) BEGIN
 5        integer x(im,jm)
 6  CSMS$DISTRIBUTE END
 7        integer xmax
 8  CSMS$PARALLEL(DECOMP_IJ,<i>,<j>) BEGIN
 9        xmax = 0
10        do 200 j=1,jm
11        do 200 i=1,im
12          xmax = max(xmax,x(i,j))
13          if (x(i,j).le.101) then
14  CSMS$PRINT_MODE(ASYNC) BEGIN
15            print *,'WARNING:  x.le.101 !!  ',x(i,j)
16  CSMS$PRINT_MODE END
17          endif
18    200 continue
19  CSMS$PARALLEL END
20
21  CSMS$PRINT_MODE(ORDERED) BEGIN
22  CSMS$INSERT      print *,'DEBUG:  local maximum value = ',xmax
23  CSMS$PRINT_MODE END
24
25  CSMS$REDUCE(xmax,MAX)
26
27  CSMS$PRINT_MODE(ROOT) BEGIN
28        print *,'maximum value = ',xmax
29  CSMS$PRINT_MODE END
30        return
```

```
    31        end
```

Example 9-3 Subroutine showing various examples of use of SMS print modes.

Assume the executable is called *print_modes*. When the serial code versions of *binary_io* and then *print_modes* are run, the following is printed on the screen:

```
>> binary_io
>> print_modes

 WARNING:  x.le.101 !!   101
 maximum value =  1005
```

When the parallel codes are run on 1 process, the following is printed on the screen:

```
>> smsRun 1 binary_io_parallel

>> smsRun 1 print_modes_parallel
 WARNING:  x.le.101 !!   101
 DEBUG:  local maximum value =  1005
 maximum value =  1005
```

For 4 processes:

```
>> smsRun 4 print_modes_parallel
 WARNING:  x.le.101 !!   101
 DEBUG:  local maximum value =  503
 DEBUG:  local maximum value =  1003
 DEBUG:  local maximum value =  505
 DEBUG:  local maximum value =  1005
 maximum value =  1005
```

The print statement on line 28 in Example 9-3, is printed using the ROOT print mode. This mode causes a single message to be printed on the screen. Only one system-dependent designated process will execute the print statement; the others will skip it and can immediately continue with useful computations. The ROOT print mode will cause the parallel code to print the same messages as the serial code in most cases.

The print statement on line 22 is executed using the ORDERED print mode. This mode causes one message to be printed on the screen for each process and guarantees that the messages always appear in the same order. It is most useful for debugging purposes. However, in order to guarantee message ordering, no process can continue until all processes have executed the print statement. This means care must be taken that all processes will ALWAYS execute an ordered print or the program will hang. For, suppose we use the ORDERED print mode at line 14:

```
      if (x(i,j).le.101) then
CSMS$PRINT_MODE(ORDERED) BEGIN
        print *,'WARNING:  x.le.101 !!  ',x(i,j)
CSMS$PRINT_MODE END
```

```
        endif
```

In this case, we see the same results for the one-process run. However, the four-process run produces the following results:

```
>> smsRun 4 print_modes_parallel
 WARNING:  x.le.101 !!   101
 DEBUG:  local maximum value =  1003
 DEBUG:  local maximum value =  505
 DEBUG:  local maximum value =  1005
```

In this case, the program hangs (deadlocks) before the final message can be printed because the warning print statement is now an ordered-mode print that has been executed by only one process. The program will wait forever for the other processes to enter this print statement. The root mode is also not appropriate here because the warning message would not be printed if point 101 were not on the root process. In this case deadlock would not occur, but the warning message would also not be printed.

The asynchronous mode is the proper mode to use in cases like the printed warning statement on line 15 (Example 9-3) because there is no guarantee that all processes will execute the print statement. In this mode, one message will appear on the screen for each process that executes the print statement. Like the root mode, there is no process synchronization during asynchronous prints. As a result, ordering of print statements may vary from one run to the next when asynchronous mode is used. For example, suppose we use the ASYNC mode for line 22 instead of ORDERED.

```
CSMS$PRINT_MODE(ASYNC) BEGIN
CSMS$INSERT      print *,'DEBUG:  local maximum value = ',xmax
CSMS$PRINT_MODE END
```

Running with four processes two times might produce the following results:

```
>> smsRun 4 print_modes_parallel
 DEBUG:  local maximum value =  1005
 DEBUG:  local maximum value =  1003
 DEBUG:  local maximum value =  505
 WARNING:  x.le.101 !!   101
 DEBUG:  local maximum value =  503
 maximum value =  1005

>> smsRun 4 print_modes_parallel
 DEBUG:  local maximum value =  505
 DEBUG:  local maximum value =  1005
 WARNING:  x.le.101 !!   101
 DEBUG:  local maximum value =  1003
 DEBUG:  local maximum value =  503
 maximum value =  1005
```

Note that the asynchronous-mode prints can appear in any order and can even appear out-of-order with other non-asynchronous-mode prints. This can be confusing in some cases. Also important to note is that ASYNC mode does not work properly when the SMS program is being run in "serverless" mode (see Section 9.4.3). The timing of when the print output appears is unpredictable.

If we remove lines 27 and 29 then there is no specific print mode in the code. In that case, SMS uses the value of environment variable SMS_PUTS_MODE. It can be set to ROOT, ORDERED, or ASYNC. If the environment variable is not defined then it defaults to ROOT.

To implement formatted output of decomposed arrays, either the SERIAL directive can be applied or, in some cases, the approach shown in Example 9-2 can be used.

9.4 I/O Performance Tuning

This section discusses ways the user of SMS can optimize the I/O performance of their codes. These optimizations require a good understanding of how input and output operations are handled in SMS. If you wish to ignore this discussion, the following table offers suggested values for the environment variables used to tune SMS I/O.

There are two different cases:

CASE I: Input files will fit in server memory

	Server/No Cachers	Server/Cachers	Serverless
SMS_RBS	size of largest input file in bytes divided by (SMS_RBC-1)	size of largest input file in bytes divided by (SMS_RBC-1)	default
SMS_RBC	default (16)	default (16)	N/A
SMS_WBS	size of largest output file in bytes	default	default
SMS_CLOSE_MODE	require-flush	require-flush	N/A
SMS_IOC_SIZE	N/A	size of largest output File in bytes divided by the number of cache processes and multiplied by 2	N/A
SMS_RAN_RSTYLE	file	file	N/A

CASE II: Input files will NOT fit in server memory (only affects input)

	Server/No Cachers	Server/Cachers	Serverless
SMS_RBS	size of largest input variable in bytes divided by (SMS_RBC-1)	size of largest input variable in bytes divided by (SMS_RBC-1)	default
SMS_RBC	default (16)	default (16)	N/A
SMS_RAN_RSTYLE	one-var	one-var	N/A

Figure 9-4: Suggested values for SMS environment variables that affect I/O performance,

9.4.1 General Guidelines

Two general guidelines should always be considered to improve both serial and parallel I/O performance. First, the user should do as little I/O as possible. Since I/O operations do not scale well, their effect on parallel performance will increase as the number of processes increase. What is an insignificant amount of run-time for 2 processes may be quite significant for 200 processes.

One optimization that can be very useful is to optionally turn off all print statements. Many serial codes already allow users to turn off some or even all print statements by setting a flag at run-time. This may speed up the serial code in some cases. The optimization is very useful in a parallel code where, on some machines, disabling prints can result in significant performance improvements. The following code fragment illustrates this common optimization:

```
if (print_enabled) then
  print *,'whatever...'
endif
```

In this case, "print_enabled" could be input through a namelist at the beginning of program execution.

A second general guideline to improve I/O performance is to combine I/O operations whenever possible. For example,

```
read(10) u
read(10) v
```

could be combined into a single read statement:

```
read(10) u,v
```

This will maximize the size of data blocks read from or written to disk and minimize I/O latency. Both unformatted and formatted statements should be considered for these optimizations.

9.4.2 The SMS Server Process

By default, SMS designates an additional process, called the server process, to manage the other processes and to handle all formatted and unformatted I/O operations. This allows computations to be done concurrent with I/O operations and can improve the overall performance of SMS program execution. Figure 9-5 illustrates a program run using four compute processes and a SMS server process.

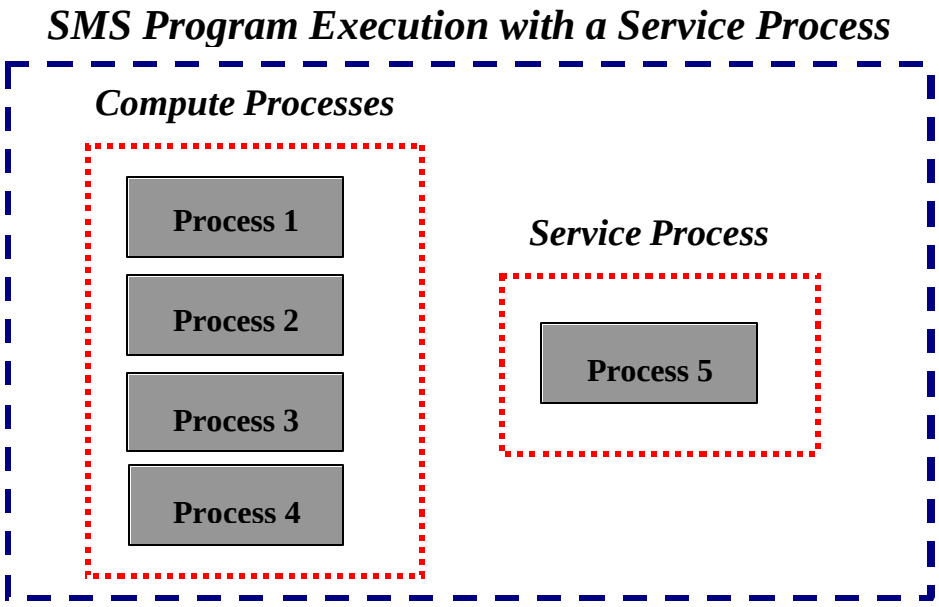


Figure 9-5: In this example, four processes are requested to run the program. By default, an additional process, called the server process, will be used by SMS for process management and I/O operations.

9.4.3 Serverless I/O

For small numbers of processors (less than 8), it may be beneficial to combine the server process functions with one of the computational processes. This type of operation is called serverless I/O and is illustrated in Figure 9-6.

If serverless I/O is used, the I/O functions that would normally be run on a separate process will be combined with one of the compute processes. Serverless SMS can be requested through an environment variable given by the command:

```
>> setenv SMS_SERVER_MODE serverless
```

On most machines, where there will be a one-to-one correspondence between processes and processors, serverless I/O will improve performance by making one more processor available to do computations. However, when large numbers of processes are used, program execution will usually be faster when a server process is used.

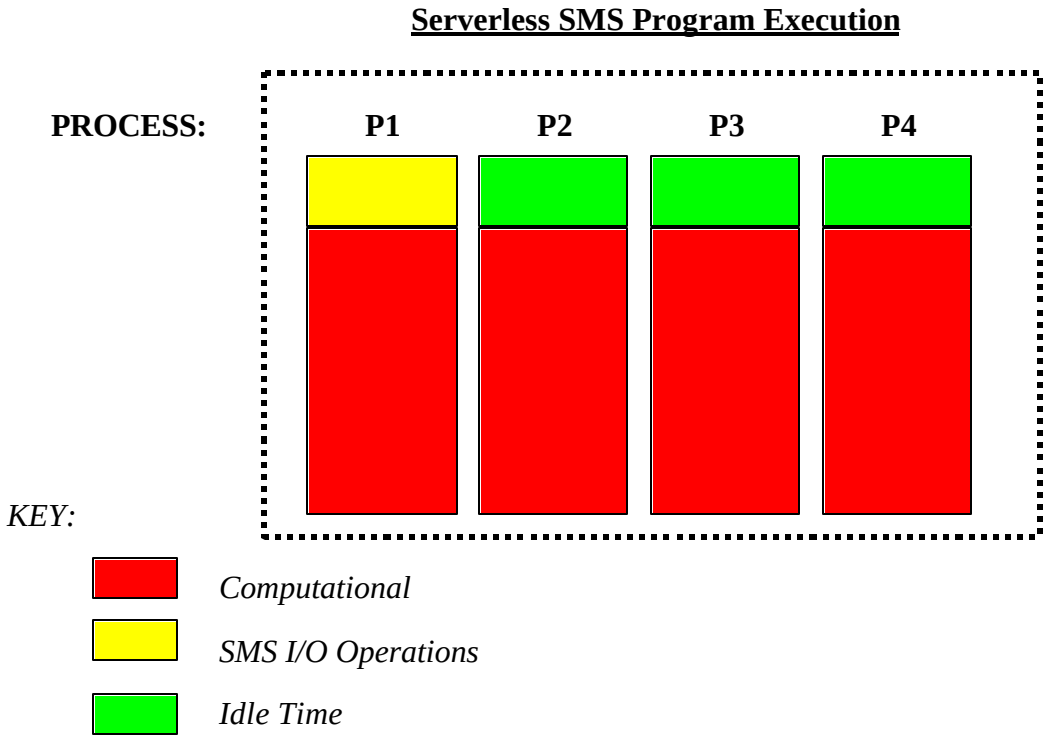


Figure 9-6: An illustration of four SMS processes used to run a program without a server process. In this example, process P1 must handle both program computations and SMS server functions that include I/O operations. While these operations occur, the other processes will be idle.

9.4.4 The FLUSH_OUTPUT Directive

The FLUSH_OUTPUT directive is used to optimize output performance; it is only useful when a server process is present. During write operations, the I/O server process buffers the data to be output in memory, re-orders the decomposed data into serial order, and then writes it out in large blocks to disk. By default, any write to disk will be delayed until the buffer is full or the file is closed. When this happens, buffers are "flushed" and their contents written to disk in large blocks. While buffers are being flushed, any processes requesting I/O services will have to wait until the flush operation is complete. The environment variable SMS_CLOSE_MODE can be set to “require-flush” for full user control of when buffers are flushed (unless they are full).

Further performance improvement can be gained by controlling when these buffers are flushed using the SMS directive, FLUSH_OUTPUT. This directive instructs the SMS I/O server process to flush the buffers immediately. If FLUSH_OUTPUT is placed so no other I/O requests are made during the flush operation, then no process will have to wait for the flush. If any I/O request is encountered, it must wait until the flush operation is complete thus minimizing the effectiveness of FLUSH_OUTPUT.

The following code fragment shows how this directive can be used:

```
      open (17,file='main_fields.dat',form='unformatted')
      write (17) u,v,w,p,t
      close (17)
c useful computations ...

      open (17,file='moisture.dat',form='unformatted')
      write (17) qs,qi,qr,qg,qw
      close (17)
CSMS$FLUSH_OUTPUT

c more useful computation ...
```

Example 9-4. Proper placement of a FLUSH_OUTPUT directive.

In this example, two files are written. As long as no other I/O (unformatted, formatted, or print) operations occur while the flush instruction is being processed, useful computations will proceed at full speed while data is simultaneously re-ordered and written to disk. This ability to overlap I/O with useful computation is key to achieving scalable I/O performance on many machines. However, any I/O statement that occurs soon after the flush operation will be sufficient to make the directive ineffective. For example, when a print statement appears just after the FLUSH_OUTPUT, it will force one of the processes to wait until the flush operation completes. Most likely, all other processes will eventually end up waiting for this process and useful computation will quickly come to a halt until the flush completes:

```
      open (17,file='diagnostics.dat',form='unformatted')
      write (17) x1,x2
      close (17)
CSMS$FLUSH_OUTPUT
      print *,'bad idea to print something here...'
... more useful computation
```

Example 9-5. Improper placement of a FLUSH_OUTPUT directive.

9.4.5 Improving Output Performance

To increase the performance of output operations, two options are available. First, SMS allows the user to designate at run-time any number of processes to serve as output cache processors. For example, Figure 9-7 illustrates a program that is run using twelve computational processes, two output cache processes, and a server process.

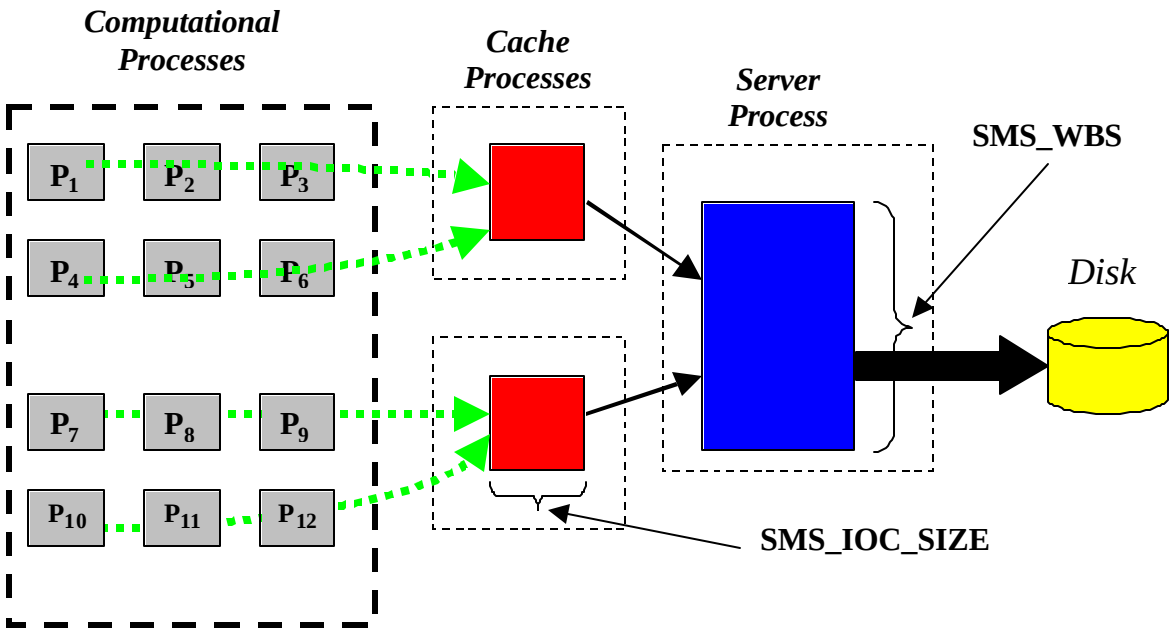


Figure 9-7: An illustration of SMS output when cache processes and a server process are used. SMS output operations pass data from the computational domain to the cache processes (if specified). Data is re-ordered on the cache processes before being passed through the server process to disk. The amount of memory allocated to the cache processes and the server process can be controlled using SMS_IOC_SIZE and SMS_WBS respectively.

The function of cache processors is to temporarily store data being output so it can be reordered and then written to disk. The computational processes can write their data to multiple cachers at high speeds and although these cachers will proceed at relatively slow speeds, total execution time is not affected because disk writes can be done at the same time as computations. Further, cache processes provide more memory capacity to temporarily store the data before it is written to disk. The number of output cachers can be requested at run-time using the -smswb option to the smsRun command. For example:

>> smsRun nprocs execname -smswb <ncachers>

executes a program where *nprocs* is the number of computational processes, *execname* is the name of the executable, and *ncachers* is the number of output cachers to be used in the run. Refer to Section 10 more details about running an SMS program.

For optimal performance, there should be enough cache processes to store all data to be output at one time. By default, SMS allocates 8 Mbytes of memory for each cache process. However, the environment variable SMS_IOC_SIZE is provided to allow the user to set the amount of memory (in bytes) they wish to allocate on each cache process. The command:

>> setenv SMS_IOC_SIZE 1000000

will allocate one million bytes of cache space. Since up to 50 percent of the cache space can be lost to the overhead required to store the data segments, a recommended size for this field is double the size of the expected output. For example, assume we wish to output the following array

```
real*4 big_array(100,200,300)
```

It will require 24 Mbytes of memory to output this array (4*100*200*300). This figure should then be doubled to account for SMS overhead costs. If each cache processor contains 10 Mbytes of memory available for SMS caching, we will need to allocate five cache processes to output this array efficiently.

A second way to improve output performance is to change the memory allocated to store data before being written to disk. Since output is always written to a buffer on the server process, modifying its size can improve performance. By default the size of this buffer is 256 Mbytes, however this value can be changed through the SMS environment variable: SMS_WBS. If write cachers are not used, then this variable should be set to the size of the largest output file when possible, otherwise output performance could degrade. When write cachers are used, the default value is usually sufficient.

9.4.6 Improving Input Performance

The server process is used to read all formatted and unformatted input data; cache processes are not used for input. If the data is decomposed, they are scattered to the other processes; if the data is non-decomposed, it is copied to the other processes.

By default, three environment variables can be used to control SMS input performance: SMS_RBS, SMS_RBC, and SMS_RAN_RSTYLE. SMS_RBS determines the size of each block that will be allocated to store input variables read from disk. SMS_RBC defines the number of blocks of size SMS_RBS that will be used for input. Finally, SMS_RAN_STYLE determines if files or individual variables will be input at one time. Figure 9-8 illustrates how these variables are used for input operations.

If a single file is input, the environment variable SMS_RBS should be set to the size of that file and SMS_RBC should be set to one. If multiple files (e.g. Initial conditions and boundary conditions) are input with differing sizes, SMS_RBS should be set to a common factor of the size of each input file. For example suppose two files are required; an initial conditions file of size 53 Mbytes and a boundary conditions of size 16 Mbytes. An approximate common factor for these two files is 8 Mbytes (8*2=16, 8*6=54). Therefore, good starting values would be: SMS_RBS=8Mbytes, SMS_RBC=6.

Using these variables, the total size of each input file should be considered when optimizing for performance. For example the execution of a program may be handled with two files: input of

initial conditions, followed by the input of boundary conditions. There should be sufficient memory on a single process to store the entire contents of each input file.

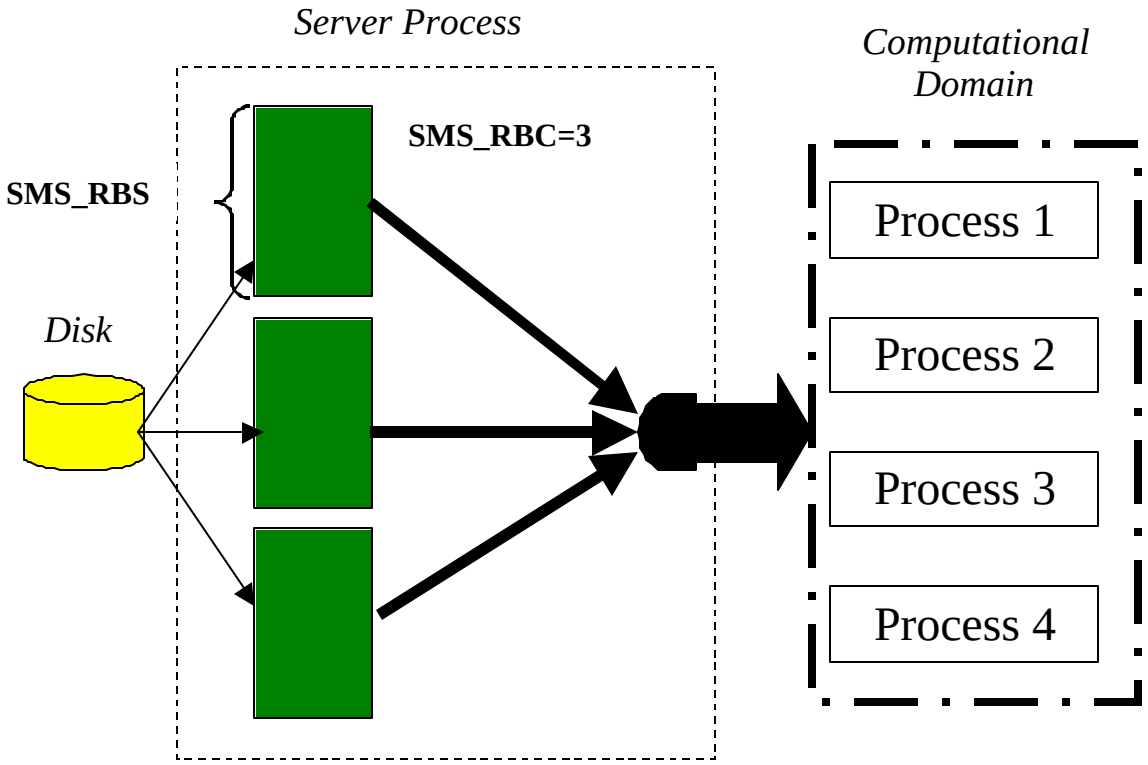


Figure 9-8: All input will pass from disk, through the server process, to individual processes within the computational domain. Two SMS environment variables can be set to control the size of two data structures within the server process: the number of input buffers (SMS_RBC) and the size of each buffer (SMS_RBS).

If not enough memory is available to store all input on a single process, SMS_RAN_RSTYLE should be set to "one-var". This will force SMS to read each variable into a buffer that resides on the server process, transfer that data to the server process for distribution among the compute processes, and then read the next variable. In this case, the quantity: SMS_RBS * (SMS_RBC-1) should be set to the size of the largest input variable.

The techniques described above are useful for reducing execution time when performance analysis indicates that run-time is limited by I/O time. Exact values of environment variables and number of cache processes are best determined by experimentation.

10 Program Termination

Parallel programs using the SMS run-time system require special handling to ensure all processes exit normally. An SMS control process is often used to manage all child processes that have been spawned through the smsRun command to execute a program. Two types of program termination are supported by SMS: a normal exit and an abort. When a program exits normally, the SMS control process will wait until every processes' computations, communications and I/O are complete before exiting. A program abort will not guarantee the completion of outstanding operations or an orderly termination of processes.

10.1 Automatic Code Generation for Termination

By default, PPP will automatically generate code to abort whenever a Fortran "stop" statement is encountered. PPP will also generate a normal exit whenever a program "end" statement is encountered. Consider the following program:

```
program main

do ii=0, num_iter
  call time_steps(ii,status)
  if (status .eq.  ABORT) then
    print *,' Model Run failed at iteration: ',ii
    stop
  endif
enddo

print *,' Model Run Successfully Completed'
stop
end
```

Example 10-1. Automatic Code Generation by PPP will cause this program to always abort.

Since the Fortran "stop" appears before the line before the end program statement, PPP will generate code to abort the parallel run. During code translation the following warning message will always appear when source contains a fortran stop statement:

```
WARNING: Program abort detected.
```

Since the intent of the original code in this case is to exit normally from the program, two actions can be taken to ensure this happens in the PPP generated source. Either the second "stop" statement (above the "end") should be removed, or the EXIT directive should be used as illustrated in the next section.

10.2 EXIT Directive

EXIT is used to control the run-time behavior of an SMS program. This directive, when inserted just before a "stop" statement, will instruct PPP to generate code to exit rather than abort. The proper placement of this directive is illustrated in Example 10-2 below:

```
      program main

      do ii=0, num_iter
        call time_steps(ii,status)
        if (status .eq. ABORT) then
          print *,' Model Run failed at iteration: ',ii
          stop
        endif
      enddo

      print *,' Model Run Successfully Completed'
CSMS$EXIT
      stop
      end
```

Example 10-2. Using CSMS$EXIT to override automatic translations

In this example, a PPP warning message will automatically be generated for every stop statement that is not immediately preceeded by the EXIT directive.

10.3 MESSAGE Directive

MESSAGE, is used to send a message to the user at run-time and optionally terminate execution of the program when it is encountered. This directive is useful when the user wishes to avoid unnecessary parallelization of code they believe is never executed. Three run-time actions are available to the user of MESSAGE: ABORT, terminates execution after writing the given message to stderr, WARN writes the given text to stderr, and INFORM writes the text to stdout.

```
      if (condition_ever_met) then
CSMS$MESSAGE(ABORT,'COMPS: THIS CODE HAS NOT BEEN PARALLELIZED BY SMS')
          call comps(a,b,c,d,NX,NY)
      endif
```

Example 10-3. Using MESSAGE to output run-time messages.

In this example, the programmer believes the subroutine *comps* is never executed so rather than parallelizing it, MESSAGE is used. Since ABORT is specified, SMS will terminate the execution of this program after the message is output to stderr.

11 Building a Parallel Program

11.1 Overview

This section describes how to use the Parallel Pre-processor (PPP) to translate Fortran code into SMS parallel source. Output files, named automatically by PPP, will be introduced in Section 11.2. Several command line options to PPP are described in Section 11.3. In Section 11.4, a simple makefile is described which can be used to build a serial or SMS parallel code. In addition, various relevant compiler and linker options are discussed in this section. Building parallel source using PPP can result in both syntactic and semantic errors that must be corrected. Section 11.5 will discuss how to interpret these PPP generated messages. Finally, Section 11.6 will describe compiler errors due to namespace conflicts from PPP generated source.

11.2 PPP Generated Output Files

Output files generated by PPP are named automatically. Include files will be named by appending ".SMS" to the original file name (e.g. `params.h` becomes `params.h.SMS`). All other source files will be named by appending "_sms" to the body of the original filename (e.g. `main.f` becomes `main_sms.f`). Intermediate files are also generated during the code translation process. These files, appended with the suffix ".tmp", remain after PPP translation. When errors are detected in the code during code parallelization, PPP messages will be generated that reference these intermediate files (see Example 11-6). Any corrections should still go into the original file from which translated code is generated by PPP.

11.3 Building SMS Parallel Source Code

The transformation of Fortran code into parallel SMS code requires the use of PPP. PPP translations are based on both its analysis of the original code and the SMS directives that were inserted into the code. This section describes how to use PPP to create parallel code at the command line, defines what code generation options are available, and gives some examples.

11.3.1 PPP Command Line Options

All PPP code translations are managed through a command line script called **ppp**. A single file can be processed at a time and no inter-procedural analysis is done. PPP is invoked by: **ppp [options] filename**. Command line options currently available are:

--checkfirst	A useful optimization to avoid PPP processing of files that do not require translation. This option can be used to allow more flexible use of suffix rules (see Section 11.4). If no I/O statements or directives are found, no PPP processing is done and the following message is output:

File has no directives - SKIPPING PPP PROCESSING

--comment	leaves replaced lines in the code as Fortran comments. This can be useful for debugging the parallel code. Note: the string used to comment out the original code is **C-PPP**.
--ExtendedSource	allow valid Fortran source to extend beyond 72 characters
--Fcommon	name of an optional include file that is not part of the original source code. Typically it will contain data decomposition directives (see Example 11-4)
--Finclude	name of an included file to be parallelized that is referenced in the source file being translated by PPP (see Example 11-2)
--Fvisible	file(s) to be made visible to PPP in order to correctly translate the current file. This option is only required for a series of nested include files (see Example 11-3)
--header	indicates the type of file to be translated is a Fortran include file
--help	prints the command line options
--IncludePath	include file search path. Similar to -I F77/F90 compiler option
--Verbose	controls the output of PPP diagnostic and code analysis messages. Errors, Warnings and Notes are output based on the verbose value. (see Example 11-7).

11.3.2 Examples

Example 11-1 shows how to build a parallel version of an include file:

>> ppp --header params.h

```
[params.h]

      parameter(nx=50, ny=50)

CSMS$DECLARE_DECOMP(decomp, <nx, ny>)

C     global variable declarations …
```

Example 11-1. Building any Fortran include file requires the --header option.

Example 11-2 shows how to use the parallel version of an include file when translating an executable code file. Since the translation of *params.h* will result in an SMS parallel version of this file (*params.h.SMS*), we use the --Finclude option to ensure this include file reference will be changed in the parallel version of *dynamics.f*.

>> ppp --Finclude=params.h --comment dynamics.f

[dynamics.f]

```
                          ..........................................................
      program dynamics

      include 'params.h'

c   Fortran code ...

      end

           ..........................................................
                       GENERATED PARALLEL PSEUDO CODE
           ..........................................................
```

[dynamics_sms.f]

```
      program dynamics

C-PPP      include 'params.h'
      include 'params.h.SMS'

c    Fortran code

      end
```

Example 11-2: The –Finclude option is used to specify the Fortran include file params.h which is referenced in the file (dynamics.f) being translated. This ensures the parallel (translated) include file will be referenced in the translated output of dynamics.f.

Example 11-3 illustrates the use of the --Fvisible option. In this example, the file "variables.h" requires information about the data decompositions listed in "params.h" to correctly translate the declarations "a" and "b" enclosed within the DISTRIBUTE directive. In particular, the array dimensions *nx, ny* and *nz* must be translated to process local sizes using information provided by DECLARE_DECOMP. The --Fvisible option is used is used to make params.h "visible" to variables.h.

>> ppp --header params.h
>> ppp --Fvisible=params.h --header variables.h

>> ppp --Finclude=params.h --Finclude=variables.h main.f

```
[params.h]
.....................................................
      parameter(nx=50, ny=50)
CSMS$DECLARE_DECOMP(decomp, nx, ny)

C     global variable declarations ...
.....................................................

[variables.h]
.....................................................
CSMS$DISTRIBUTE(decomp, nx, ny) BEGIN
      real a(nx, ny, nz)
      real b(nx, ny, nz)
CSMS$DISTRIBUTE END
.....................................................


[main.f]
.....................................................
      program main

      include 'params.h'
      include 'variables.h'

c     other code ...

      end
```

Example 11-3: The --Fvisible option is used when inter-dependent include files must be translated.

In Example 11-1, the CSMS$DECLARE_DECOMP was added to an include file that already existed (params.h). If the user prefers to insert the SMS directives into a separate "directives" file, the option --Fcommon is used instead of --Finclude. Example 11-4 illustrates the --Fcommon option.

>> ppp --header sms.inc
>> ppp --Fcommon=directives.inc dynamics.f

```
[directives.inc]
.....................................................
      parameter(nx=50, ny=50)
CSMS$DECLARE_DECOMP(decomp,<nx,ny>)
.....................................................


[dynamics.f]
.....................................................
```

```
      program main

      include 'params.h'

c           more Fortran code ...

      end

          ....................................................
                        GENERATED PARALLEL PSEUDO CODE
          ....................................................
      program main

      include 'directives.inc.SMS'
      include 'params.h'

c     more Fortran code ...

      end
```

Example 11-4: In this example DECLARE_DECOMP, defined in "directives.inc", is referenced (and required) by "dynamics.f". Note: Since params.h no longer contains any SMS directives and will not be translated by PPP, it CANNOT be listed using the -Finclude command line option.

11.4 Building PPP Executables

A simple makefile is presented to aid the user in translating their sequential codes into SMS codes. This file assumes the variable "SMS" has been set to the location where the SMS software has been installed. This can either be set explicitly in the Makefile at line 5, or defined as an environment variable (e.g. setenv SMS *pathname*).

```
 1    # standard make file used to build serial or SMS parallel executables
 2
 3    .SUFFIXES:  .s .p
 4    #
 5    SMS = /usr/local/sms
 6
 7    # system specific compilation flags (for an SGI Origin 2000)
 8    COMPILER = f77
 9    COMP_FLAGS = -O2 -64 -mips4 -r10000 -fixedform -I$(SMS)/include
10
11    #     SMS link libraries
12    LIBS = -L$(SMS)/lib -lfnnt -lnnt -lsrs -lppp_support -lmpi
13
14    #     PPP specific options set here
15    PPP = $(SMS)/bin/ppp
16    PPP_FLAGS = --Finclude=params.h --Finclude=variables.h --comment \
17    --checkfirst
18    PPP_HEADER_FLAGS = --header --comment
```

```
19
20    #     include files
21    INCLUDES = params.h variables.h globals.h
22    PINCLUDES = ${INCLUDES:.h=.H}
23
24    #     object files
25    OBJS = file1.o file2.o file3.o
26
27    PFILES = ${OBJS:.o=.p}
28    SFILES = ${OBJS:.o=.s}
29
30    #     executable target names
31    parallel:   $(PINCLUDES) $(PFILES)
32          $(COMPILER) -o par_prog $(OBJS) $(COMP_FLAGS) $(LIBS)
33
34    serial:     $(INCLUDES) $(SFILES)
35                $(COMPILER) -o seq_prog $(OBJS) $(COMP_FLAGS) $(LIBS)
36
37    #     suffix rules for sequential and parallel source
38    .f.s:       $(INCLUDES)
39          $(COMPILER) -c $(COMP_FLAGS) $<
40
41    .f.p:       $(PINCLUDES)
42                $(PPP) $(PPP_FLAGS) $*.p
43                $(COMPILER) -c $(COMP_FLAGS) $*_sms.f
44                mv $*_sms.o $*.o
45
46    # include file translations
47    params.H:   params.h
48                $(PPP) $(PPP_HEADER_FLAGS) params.h
49
50    variables.H:      variables.h params.h
51                $(PPP) $(PPP_HEADER_FLAGS) --Fvisible=params.h variables.h
52
53    globals.H:
54
55    clean:
56          /bin/rm *_sms.f *.SMS *.o *.tmp
```

Example 11-5. A makefile for serial or parallel source.

11.4.1 Makefile Compiler and Linker Options

The Fortran compiler flags (COMP_FLAGS on line 9) are set for an SGI Origin 2000. Other systems will require different options. A makefile provided in the SMS distribution ($SMS/lib/makefile.header) gives recommended compilation flags (found in variable STD_OPT_FLAGS) that should be used when modifying COMP_FLAGS for the target machine.

11.4.2 Include File Handling

Include files are listed for both parallel and sequential source in the makefile variable INCLUDES. Parallel include files (line 22) are translated using SMS are built using the explicit

targets *params.H* and *variables.H* (lines 47-51*)*. Notice the PPP command to build *variables.h* (line 51) contains the --Fvisible option in addition to the standard ppp flags defined by: PPP_HEADER_FLAGS at line 18. Since *variables.h* requires information from *params.h* for proper translation, this option is required (see Example 11-3).

PPP_FLAGS (lines 16-17) lists the include files that are translated by PPP via the –Finclude option. This option is required to ensure any references to these files in Fortran source will be modified to their parallel filename (see
Example 11-2).

11.4.3 Building the Object Files

Two suffix rules are used to build sequential or parallel object source. Sequential source files are built using the first ***(.f.s)*** suffix rule (line 38) while parallel source rely on the second ***(.f.p)*** suffix rule (line 41). This makefile uses ***.s*** for serial and ***.p*** for parallel but any suffix name could have been used. Using these rules to build an SMS parallel object file from the file ***`file1.f`***, for example, the user would enter:

```
>> make file1.p
```

PPP generated source is written to the file: ***`file1_sms.f`***, and the object file: ***`file1.o`*** would be built unless compilation errors occurred.

Similarly, to build a serial object file, the user would enter:

```
>> make file1.s
```

11.4.4 Building the Executable

In addition to building single object files, this makefile can also build a parallel or serial executable from a set of object files. Using a pre-defined list of object file names (OBJS on line 25)), parallel (PFILES at 27) and serial (SFILES at line 28) files are determined and listed as dependencies for each target executable. This assumes there is a direct mapping between the object and source file names (e.g. file1.o maps to file1.f; not something else).

Then to build the SMS parallel executable “par_prog” in this makefile, the user would enter:

>> make parallel

Similarly, the user would enter the following to build a serial executable called seq_prog:

>> make serial

11.5 PPP Error Reporting

Two types of errors are reported by PPP: parsing errors and semantic errors. Parsing errors must be corrected before further translations of the input file are permitted. Semantic errors are reported as errors, warnings or notes. These messages can be controlled through the --verbose option of PPP discussed in Section 11.5.2.

11.5.1 Parsing Errors

Parsing errors occur when PPP cannot resolve the Fortran code to the grammar defined by the SMS/PPP directives (refer to the SMS Reference Manual), and the Fortran 77 language. Further details about language extensions supported by SMS can be found at:

http://www-ad.fsl.noaa.gov/ac/SMS_Supported_Fortran_Features.html

The parser currently supports statements or PPP directives that are up to 500 characters in length. Multiple statement lines are collapsed and white space is removed before statements are parsed. Statements longer than 500 characters will not be parsed correctly in PPP.

The form of a parsing error message is:

<filename> <line> <column> <error type> <message>

filename	- name of file being parsed
line	- line number
column	- column number in which error occurred
error type	- types are: **ERROR, WARNING, NOTE**
message	- diagnostic message

An example of a PPP generated parsing error is shown in Example 11-6.

```
 1 CSMS$DECLARE_DECOMP(spec_dh,<jtrun>)
 2 CSMS$DISTRIBUTE(spec_dh, jtrun) BEGIN
 3       real*8 cc(jtrun), bb(jtrun)
 4 CSMS$DISTRIBUTE END
 5
 6 CSMS$PARALLEL(spec_dh, m) BEGIN
 7       do 3 m=2, jtrun, 2
 8         cc(m) = cc(m) + bb(m)
 9       continue
10
11    C           CSMS$PARALLEL END is missing
12
13       end
```

Example 11-6. Code that generates a PPP parsing error.

PPP generates the following error **mes**sage:

```
“Loops_sms.f.tmp" 13 501 ERROR: Syntax error
“Loops_sms.f.tmp" 13 501 NOTE Parsing resumed here
```

This message indicates the parser failed in the file *Loops_sms.f.tmp* at line 13 column 501. A parsing error occurring at column 501 indicates no resolution of the statement to the grammar by the end of the line. In the example, the parser expects a PARALLEL END directive before the end of the file. Naturally, the error should be corrected in the original file (*Loops.f*) rather than the PPP generated file.

11.5.2 Semantic Errors

Semantic errors are reported when a section of code targeted for translation has an error (a PPP ERROR), may cause incorrect code to be generated (a PPP WARNING), or identifies a place where a particular type of transformation occurred or PPP language limitation was detected (a PPP NOTE). By default, all PPP ERROR messages will be output. Control of semantic errors are handled through the PPP command line option: --verbose = <value>. Four verbose options are permitted:

value	message domain
0	no semantic messages are output (not recommended)
1	PPP ERRORS only (DEFAULT)
2	PPP ERRORS and WARNINGS only
3	PPP ERRORS, WARNINGS and NOTES

While the error messages should always be addressed, warning messages may also be useful for detecting potential problems. For example, the code segment in Example 11-7 below causes PPP to generate the following important warning message:

```
./IO.f.tmp” 11 13 WARNING: This variable, decomposed by CSMS$DISTRIBUTE, is
being used outside of a parallel region.
```

This warning message indicates a problem on line 11, column 13 of the PPP generated file *IO.f.tmp*. The variable *cc* was defined to be a distributed array (using DISTRIBUTE) but is being referenced outside a parallel region (PARALLEL). Further explanation on the use of these directives can be found in Section 2.3.

```
>> ppp --verbose=2 IO.p
```

```
 1    CSMS$DISTRIBUTE(dh, m, n) BEGIN
 2          real cc(m,n)
 3    CSMS$DISTRIBUTE END

 4          do i = 1, m
 5            do j = 1, n
 6              cc(i,j) = 0.0
 7            enddo
 8          enddo
 9
10    c   more code ...
```

Example 11-7. Code that generates a WARNING because the decomposed variable "cc" is being used outside of a parallel region.

11.6 Compilation Errors

During the parallelization process PPP generates new variables for some translations. PPP variables are either automatically generated or defined explicitly by PPP. Explicitly defined names will always contain a double underscore in their name (e.g. ppp__status). To avoid compiler errors due to name space conflicts, avoid using variable names with double underscores in them. For example, the sequential code cannot contain a variable called `PPP__status` because PPP translation explicitly defines another variable called `ppp__status` for its own use. A compilation error would result because two variables would be declared with the same name.

12 Running a SMS Program

12.1 Introduction

Once a program has been translated into SMS parallel code (Section 11.3) and linked to the appropriate libraries (see Section 11.4), it can be run on one or more processors using the SMS run-time executable smsRun. The syntax for smsRun is:

```
>> smsRun numprocs execname [options]
```

By default, SMS uses an additional server process to perform I/O operations, and provide overall management and control services for the other processes. For example, to run the executable test with two processes and one server process, the user would enter:

```
>> smsRun 2 test
```

It is possible to take advantage of the idle compute cycles available on the server process by setting SMS environment variable SMS_SERVER_MODE to *`serverless`*. This will permit computational and management functions to co-exist in a single process. This option is beneficial when only a small number of processors are available. However, as the numbers of processes grow, the cost of performing both server functions and computations will limit the performance of the other dependent processes.

Figure 9-5 assumes a single process is run on each processor. However, SMS permits the user to request more processes (using smsRun) than available processors. For example if my_program was run with 20 processes:

>> smsRun 20 my_program

on a system with only 16 processors, five processors would contain two processes, one would contain the server process, and the rest would each contain a single process designated to run the program. This is a bad idea because performance will suffer whenever multiple processes are scheduled on a single processor on most machines.

12.2 Optional Command Line Arguments

Several optional arguments to smsRun are permitted. One optional argument to control the number of I/O write-cache processes to be dedicated to the program's execution can be expressed by:

```
>> smsRun numprocs execname -smswc numcacheprocs
```

The use of write-cache processes to improve performance is discussed in Section 9.4.5. Another option, -sms-, allows the user to specify machine specific arguments to the underlying

communication layer (e.g. MPI, SHMEM) directly. All arguments that follow this option will be ignored by SMS and passed directly to the communications software. For example:

>> smsRun 3 test -sms- -mpi_special

illustrates a way to pass the run-time option -mpi_special to the underlying MPI executable (mpirun) to specify node names on a network of work stations. Information about other machine specific optins for smsRun are available at the following SMS web site:

http://www-ad.fsl.noaa.gov/ac/SMS_Run_Options.html

12.3 Run-time Environment Variables

Several environment variables can also be set to control the run-time behavior of SMS. The following environment variables are available:

SMS_BITWISE	Set to "EXACT" to use bit-wise exact reductions - see Section 7.2
SMS_CHECK_HALO	Set to "ON" to execute checks of halo regions specified by CHECK_HALO directives.
SMS_CLOSE_MODE	
SMS_IO_FORMAT	Used to specify file format for files that are read or written by SMS (see Section 9.1).
SMS_IOC_SIZE	Improving Output Performance (see page 125)
SMS_PUTS_MODE	Modifies the default behavior of formatted output. Options are: ROOT, ASYNC and ORDERED. See Section 9.3 for more details about these options.
SMS_RAN_RSTYLE	Improving Input Performance (see page 127)
SMS_RBC	Improving Input Performance (see page 127)
SMS_RBS	Improving Input Performance (see page 127)
SMS_READ_FORMAT	Used to specify file format for files that are read by SMS (see Section 9.1).
SMS_SERVER_MODE	The SMS Server Process (see page 122)
SMS_TIMER_LEVEL	
SMS_WBS	See Section 9.4.5 - page 125
SMS_WRITE_FORMAT	Used to specify file format for files that are written by SMS (see Section 9.1).
SMS_XFERMODE	Controls transfer algorithms that are used to implement TRANSFER. Options are: "logn" and "original"

12.4 Run-time Error Messages

When an error occurs in an SMS program, execution will usually terminate and SMS will generate an informational message describing the source file name, line number, and a brief summary of the problem. A complete set of SMS run-time error messages is available at the following SMS web site:

http://www-ad.fsl.noaa.gov/ac/SMS_Messages.html

Example 12-1 illustrates SMS run-time message capabilities. Recall that the user is responsible for determining the correct number of processes over which to run the program. For static memory allocated programs, the minimum number of processes will be determined by declared local size values in DECLARE_DECOMP as discussed in Section 3.3.

```
 1       program example1
 2
 3       parameter(nx=50, ny=50)
 4       parameter(nx_a=nx/2, ny_a=ny/2)
 5 CSMS$DECLARE_DECOMP(decomp, <nx_a, ny_a>)
 6 CSMS$DISTRIBUTE(decomp, nx,ny ) BEGIN
 7       real a(nx,ny)
 8 CSMS$DISTRIBUTE END
 9
10 CSMS$CREATE_DECOMP(decomp,<nx,ny>,<0,0>)
```

>> smsRun 1 example1

```
Process:   0 Error at: ./example_sms.f.tmp:17.1
Process:   0 Error status =  -2202 MSG: DECOMPOSED ARRAYS ARE TOO SMALL.
Process:   0 Aborting...
```

Example 12-1. Code and command that generates a run-time SMS error.

After PPP translation, the array ***a*** will be defined with the declared local sizes ***nx_a*** and ***ny_a*** given in DECLARE_DECOMP. Since the local sizes of this array are half the size of the original code (nx and ny respectively), the minimum number of processes the user can run this problem is four (two in each direction). If you attempt to run on fewer processes, the program will halt with the given error message.

The first line of the error message indicates the file name and location within the file where the problem occurred. PPP generated code frequently uses sub-numbering due to multiple generated calls to SMS routines that stem from the same line of original code. In this example, a run-time error was detected by SMS at line 17 in code generated by the directive CREATE_DECOMP that can be found in temporary file: ***example1_sms.f.tmp*** (not shown).

The second line gives the SMS error message. The error messages reflects the incorrect sizing of the decomposition ***decomp***, declared by DECLARE_DECOMP and initialized by CREATE_DECOMP.

Once the problem is understood corrections to the code can be made. These corrections should go into the original file (in this case example1.f) not in the temporary file where the problem was detected and probably diagnosed. Once changes are made, ppp can be executed to re-translate the input file from which a fresh executable can be built and tested.

Appendix A: Assignment of Processes to Decomposed Dimensions

The assignment of processes to decomposed dimensions by SMS depends on the number of processes and the global sizes of the decomposed dimensions. Below are the rules that SMS follows when deciding how to allocate processes among one or two decomposed dimensions. Assume that Np = number of processes and Nd = number of decomposed dimensions:

1) If Nd=1, assign all processes to the single decomposed dimension.

2) If Np is prime, assign all processes to the decomposed dimension with the largest size.

3) If Np is not prime and Nd=2, factor Np into f1*f2 = Np, such that factors f1 and f2 are as close together as possible.

3a) If factors are equal (f1=f2), assign f1 processes to each decomposed dimension.

3b) If factors are not equal, assign a number of processes equal to the largest factor to the decomposed dimension with the largest size.

3c) If factors are not equal and sizes of decomposed dimensions are equal, assign a number of processes equal to the largest factor to the last decomposed dimension.

These rules are intended to allow for optimum performance with minimal input from the user. Rule 1 handles the simple cases where more than one decomposed dimension has been specified but only one can actually be decomposed because the number of processes available is prime. Assignment of all processes to the largest decomposed dimension will usually result in the most efficient distribution of work. Rule 3 restricts factoring of Np to keep the number of processes assigned to each dimension as close together as possible. For example, with Nd=2, 100 processes would be factored into 10*10, not 20*5 or 25*4. The effect of this rule is to keep the virtual process array as "square" as possible which can be beneficial for "exchange" type communications on some machines. Rule 3b will

cause more processes to be assigned to larger dimensions in cases where factors are not equal. This was the case in the 8-process run where im was greater than jm in the examples above. The purpose of this rule is to allow SMS to attempt to "fit" the virtual process array to the Fortran arrays as closely as possible. Rule 3c causes SMS to assign more processes to the last decomposed dimension to allow the user to control whether more processes will be assigned to the outer or inner array dimensions.

To further illustrate rule 3c, consider the following code fragments:

```
[ Fragment 1.]
CSMS$CREATE_DECOMP(DECOMP_1, <nx, ny>, <0,0>)
...
CSMS$DISTRIBUTE(DECOMP_1, <nx>, <ny>) BEGIN
      real u(nx,ny)
CSMS$DISTRIBUTE END

[ Fragment 2.]
CSMS$CREATE_DECOMP(DECOMP_2, <jm, im>, <0,0>)
...
CSMS$DISTRIBUTE(DECOMP_2, <im>, <jm>) BEGIN
      real a(im,jm)
CSMS$DISTRIBUTE END
```

In fragment 1, when *nx.EQ.ny*, more processes will be assigned to the second decomposed dimension, *ny*, which is the outer dimension of array *u*. This will preserve the longest possible vector lengths because the inner dimension of *u* (*nx*) will not be split up among the processes. This approach is good for a machine with vector processes. In fragment 2, when *im.EQ.jm*, more processes will be assigned to the second decomposed dimension, *im*, which is the inner dimension of array *a*. In some special cases, this may result in better performance on some cache-based machines. In general, the user can simply keep the decomposed dimensions in the same order as the array dimensions and expect the best performance in most cases on most machines.